Special Forth Issue

SOFTWARE TOOLS FOR ADVANCED PROGRAMMERS

Dr. Dobb's Journal

#95 September 1984

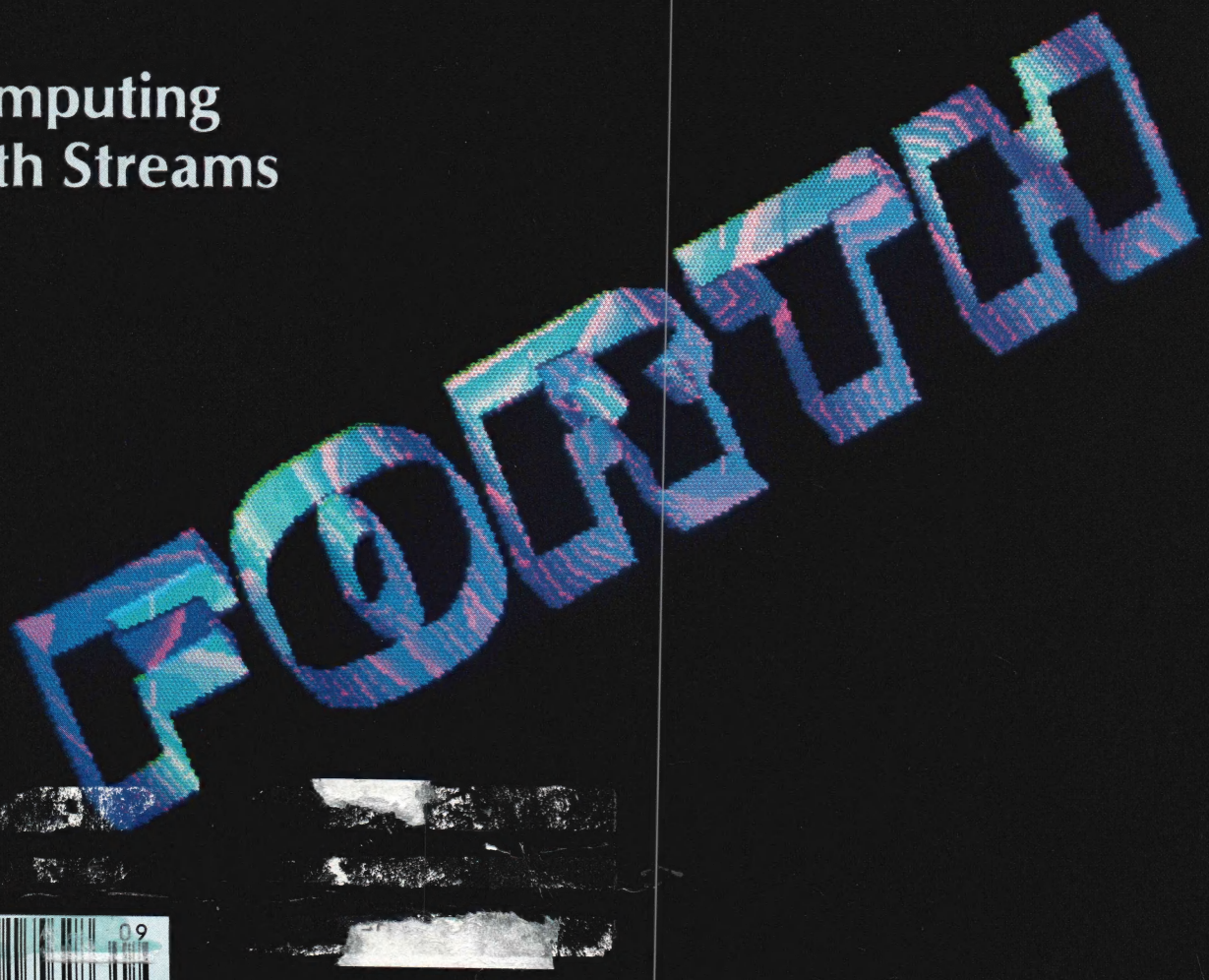
$2.95 (3.50 Canada)

Forth Cross Compiler for the MC68000

File Maintainance in Forth

Forth and the Fast Fourier Transform

Computing With Streams



0 38351 16562 09

ROCKWELL SEMICONDUCTOR TECHNOLOGY PUTS FORTH™ IN CONTROL FOR YOU.

R65F11's or R65F12's bring "Control" a two-chip solution.

Putting FORTH's high-level language to work means working 10 times faster than BASIC—and gives you a minimum design/development time. It also allows bigger programs in the same space...then running them faster.

Our R65F11 has internal ROM containing 133 run-time FORTH functions. The 8K-byte R65FR1 development ROM has 153 more functions that aid in program development. You don't need the development ROM in production systems; you just add your application program in ROM or EPROM.

FORTH is flexible, expandable and available **now** from Rockwell.

FORTH is a trademark of FORTH, Inc.

So let our R65F11 or R65F12 (with 24 more I/O lines) put you in control. Call your local Rockwell distributor or sales representative today for more information.

Rockwell Semiconductor Products Division

Rockwell International, P.O. Box C,
MS 501-300, Newport Beach, CA 92660.
Call Toll Free (800) 854-8099.
In California (800) 422-4230.

Rockwell International

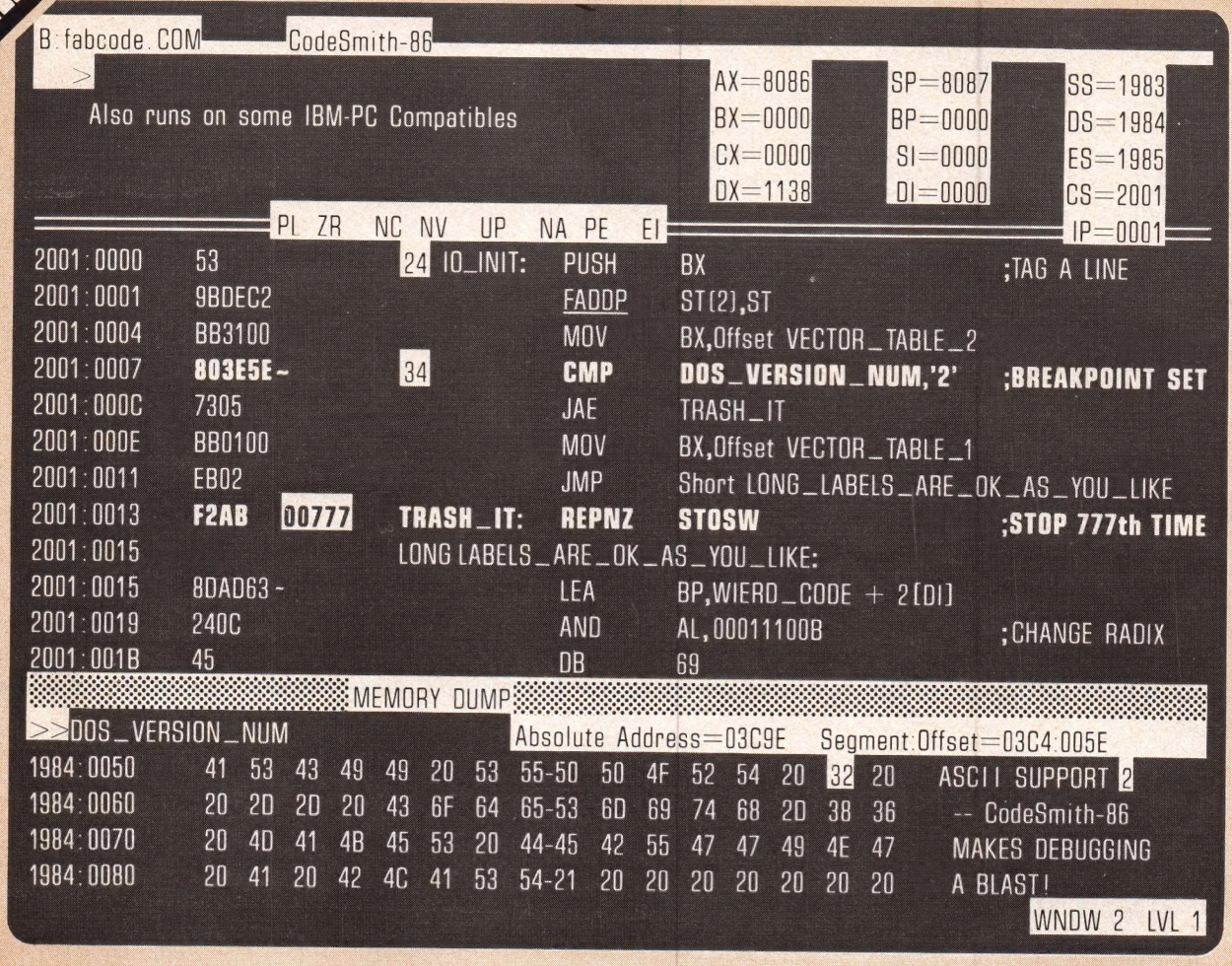
...where science gets down to business

Circle **no. 73** on reader service card.

IBM-PC SYMBOLIC DEBUGGER

New Release 1.8 – SOLID GOLD

CodeSmith™-86



It's here—THE Multi-Window Interactive Debugger that's STATE-OF-THE-ART.

- Scroll Up/Down thru full-screen disassemblies & memory dumps
- **Load** and **Write** Commands much easier, more powerful than DEBUG's
- "Snapshot" a complete debugging state onto disk—resume later
- True passpoints and execution path counters
- SCREENSAVE mode saves and restores user's graphic display when breakpoint hit
- Disassemble selected ranges of memory code to disk—compatible with IBM Assembler
- Stop on data Read/Write or memory range access

Hot-Line technical support

The Professional's Choice—CodeSmith-86

Multiple copies purchased by:
Lotus Development Corp., MicroPro, VisiCorp, IBM.

Requires MS-DOS & 160K RAM.

OEM and dealer inquiries invited.

Send ___ **VISUAL AGE** CodeSmith-86 @ $145 + $4 shipping (each) • California residents add 6.5% sales tax.
642 N. Larchmont Blvd. L.A., CA 90004 • (213) 439-2414

Name _____
Company _____
Address _____
City _____ State _____ Zip _____ Phone # _____
Acct. No. _____
Exp. Date _____
☐ VISA ☐ MasterCard
☐ Payment Enclosed
☐ COD

VISUAL AGE
642 N. Larchmont Blvd. • Los Angeles, CA 90004

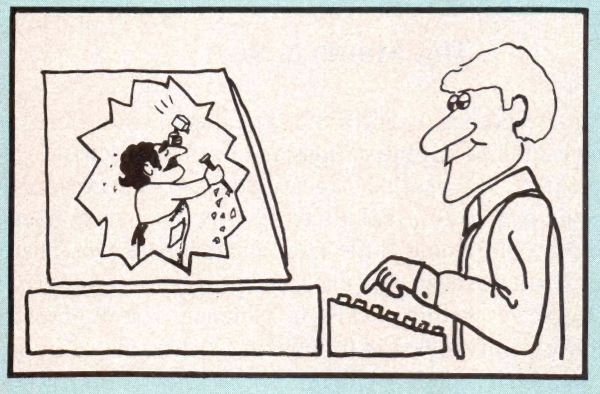
Circle **no. 93** on reader service card.

CALL (213) 439-2414

$145.00

SCREEN SCULPTOR

GENERATES CUSTOMIZED INPUT SCREEN PROGRAMS IN BASIC AND PASCAL . . .



. . . ***BECAUSE YOUR TIME IS TOO VALUABLE***

Only **$125.00**

ITEM #2010

(Includes shipping and handling)
(N.Y.S. Res. Add 8¼% sales tax)

BASIC or PASCAL. Easily!

Generate programs in **BASIC** or **PASCAL.** Your **choice.**
Works with **Turbo PASCAL * IBM * PASCAL** or **IBM BASIC** (Interpreter and Compiler).
Easy to use. Begin productive use in minutes!

POWERFUL SCREENS

Everyone can have **professional quality screens** to dress up any program.
Generate **complex, colorful, effective** screens in minutes.

FULL FUNCTION SCREEN CREATION

Simply **"draw" input screens** with word processor style editor.

Advance screen creating features include:
- **Draw boxes, lines, etc.** in seconds with unqiue character selection menu.
- **Repeat last character** in any direction.
- Special **color-select screen** displays all available colors.
- **Paint and Repaint** sections of screen at any time.
- **Copy** and **Move** sections of screen.
- **Insert** or **Delete characters** and **lines.**
- **Input field definition** screen gives you **total control** of character type definitions, edit screen masks, input sequence, variable names, initial values, protected characters, etc.

COMPLETE DATA ENTRY ROUTINES

Generates customized program code that allows **professional quality** data input using the full PC keyboard **(cursor keys, delete, insert, and more). Checks input data** for valid entries and **displays error messages.**
Programs are easily **merged** with your own programs.
Easy to follow documentation shows how and where you can **modify the generated programs.**

- **Available now with IBM PC, PCjr, PCXT, and all true compatibles.**
- **Requires 128k RAM, one floppy disk drive, and PC DOS. Works with any display type.**

* Turbo PASCAL is a registered trademark of Boreland International, Ltd. IBM is a registered trademark of IBM corporation.

Master Card and **Visa** orders call now *toll free* 1-800-824-7888. Operator 268
Alaska and Hawaii call 1-800-824-7919. Operator 268
DEALER INQUIRIES INVITED. SORRY, NO C.O.D.
Produced and Distributed by **The Software Bottling Co. of New York.**
29-14 23 Ave., New York City, N.Y. 11105 • (212) 728-2200 Telex 380748

MasterCard VISA

Circle **no. 77** on reader service card.

Dr. Dobb's Journal

Editorial
Editor-in-Chief *Michael Swaine*
Editor *Reynold Wiggins*
Managing Editor *Randy Sutherland*
Contributing Editors *Robert Blum, Dave Cortesi, Ray Duncan, Anthony Skjellum, Michael Wiesenberg*
Copy Editors *Polly Koch, Cindy Martin*
Typesetter *Jean Aring*

Production
Design Director *Fred Fehlau*
Art Director *Shelley Rae Doeden*
Production Manager *Detta Penna*
Production Assistant *Alida Hinton*
Cover *Isaac Victor Kerlow*

Advertising
Advertising Sales *Alice Hinton, Walter Andrzejewski*

Circulation
Circulation and Promotions Director *Beatrice Blatteis*
Fulfillment Manager *Stephanie Barber*
Direct Response Coordinator *Maureen Snee*
Promotions Coordinator *Jane Sharninghouse*
Single Copy Sales Coordinator *Sally Brenton*
Single Copy Sales *Lorraine McLaughlin*
Circulation Assistant *Kathleen Boyd*

M&T Publishing, Inc.
Publisher and Chairman of the Board *Otmar Weber*
Director *C.F. von Quadt*
President *Laird Foshay*

Dr. Dobb's Journal (USPS 307690) is published monthly by M&T Publishing, Inc., 2464 Embarcadero Way, Palo Alto, CA 94303, (415) 424-0600. Second class postage paid at Palo Alto and at additional entry points.

Address correction requested. Postmaster: Send Form 3579 to *Dr. Dobb's Journal*, 2464 Embarcadero Way, Palo Alto, CA 94303. **ISSN 0278-6508**

Subscription Rates: $25 per year within the United States, $44 for first class to Canada and Mexico, $62 for airmail to other countries. Payment must be in U.S. Dollars, drawn on a U.S. Bank.

Contributing Subscribers: Christine Bell, W.D. Rausch, DeWitt S. Brown, Burks A. Smith, Robert C. Luckey, Transdata Corp., Mark Ketter, Friden Mailing Equipment, Frank Lawyer, Rodney Black, Kenneth Drexler, Real Paquin, Ed Malin, John Saylor Jr., Ted A. Reuss III, InfoWorld, Stan Veit, Western Material Control, S.P. Kennedy, John Hatch, Richard Jorgensen, John Boak, Bill Spees, R.B. Sutton. **Lifetime Subscribers:** Michael S. Zick, F. Kirk.

Foreign Distributors: ASCII Publishing, Inc. (Japan), Computer Services (Australia), Computer Store (New Zealand), Computercollectief (Nederland), Homecomputer Vertriebs GMBH (West Germany), International Presse (West Germany), La Nacelle Bookstore (France), McGill's News Agency PTY LTD (Australia), Progresco (France).

People's Computer Company

Dr. Dobb's Journal is published by M&T Publishing, Inc. under license from People's Computer Company, 2682 Bishop Dr., Suite 107, San Ramon, CA 94583, a non-profit, educational corporation.

September 1984
Volume 9, Issue 9

CONTENTS

This Month & Next

With the growing acceptance of Forth-83, we expect to see more energy being put into designing useful Forth applications and less into discussion of how Forth should be implemented. This year's Forth issue reflects that expectation. A quick perusal of the facing page will reveal a Fast Fourier Transform package, a cross compiler for the MC68000, and a file maintenance package. There is also an article which uses Forth words to examine some programming issues. That the 83 Standard did not cover everything is also in evidence. You might want to examine the FVG floating-point extension to see what some folks are doing until the next meeting of the Forth Standards Team.

One theme that emerged this month, in both the FFT article and in the C/Unix Programmer's Notebook, was the use of complex numbers in programs. Those interested in more detail on the topic should stay tuned for an article next month that discusses complex numbers in greater detail. Also next month, we will feature a Unix-like generalized regular expression parser. In addition to all our regular columns, we will introduce The Software Designer, a new column focusing on a wide range of programming issues.

A Quick Fix for Subscription Bugs

By September 7, 1984, all subscribers, current and former, should have received a letter from our Circulation Department explaining that we recently contracted an excellent West Coast fulfillment service in an effort to serve you even better. We are anxious to clear up any problems you may have been having with your subscription, and we are giving you a golden opportunity to have those problems fixed. Accompanying the letter is a form for documenting any difficulties you are experiencing. If you need your address changed, your expiration date corrected, missing issues supplied, or some other adjustment, this is the best way to ensure quick action. In fact, if you have *not* received the letter yet (that may tell you something), you should call our fulfillment firm toll-free at 1-800-321-3333 (inside CA dial 1-619-485-6535); or drop Circulation a note (preferably with a magazine label) at 2464 Embarcadero Way, Palo Alto, CA 94303.

This Month's Referees

Dr. Dobb's Journal regularly draws on the expertise of a Board of Referees for technical evaluation of material submitted for publication. In addition to remarks to the editors concerning accuracy and relevance of manuscripts, the referees often provide constructive comments for authors regarding clarity or completeness. The referees who contributed to this month's issue are:

Ray Duncan, *DDJ* Contributing Editor
Kim Harris, Dysan Corporation
William Ragsdale, President, Dorado Systems

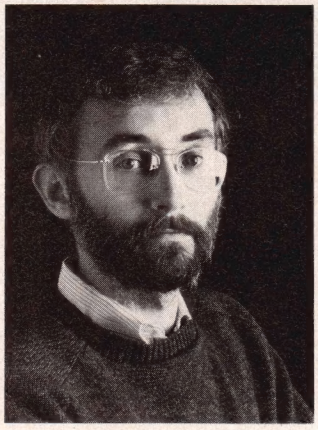
SOFTWARE TOOLS FOR ADVANCED PROGRAMMERS

Dr. Dobb's Journal

ARTICLES

DEPARTMENTS

EDITORIAL



In *A Programmer's Notebook: Utilities for CP/M-80*, our resident intern Dave Cortesi argues that programming is a craft. "Like [woodworking, pottery or blacksmithing], programming has the goal of producing utilitarian objects. Like those crafts, its practice calls for the application of skills developed by long practice, and for the application of the artisan's own good taste and imagination to the elaboration of conventional designs. And like any good craftperson, the good programmer strives to reach beyond mere utility to efficiency, elegance, and beauty."

Not everyone agrees; C.A.R. Hoare (in *IEEE Software*, April 1984) wants to claim that programming "has transcended its origins as a craft, has avoided the temptation to form itself into a priesthood, and can now be regarded as a fully-fledged engineering profession." I think Dave is closer to the truth, and that the forces pushing programming in the direction of an engineering discipline are the same forces that would maintain the priesthood. Programming as a craft is at least a good metaphor, and provides a context for discussing programming's design aspect.

The programmer's taste and imagination produce different approaches to software design, such as whether the programmer works alone or on a team, writes in C or in Ada, feels like a craftsperson or an engineer. "One does not use structural engineering analysis to build a sandcastle," Hoare says, "But neither does one choose the prize-winning builder of sandcastles as an architect for a tower block of offices in a city." Sandcastles vs skyscrapers? Hoare's distinction shows a bias toward the engineering style; I think that a more meaningful distinction is that between those problems that an individual can't solve (and that therefore require an engineering-team approach) and those that only an individual can solve. Maximizing efficiency, for example. Efficiency rewards tricky programming, and tricks are anathema to the engineering team approach.

The craft metaphor fits in other ways, too. As is typical of artisans, programmers often don't benefit from their own craft. Software designers routinely work with inadequate, ill-designed tools. Hoare would applaud "when a software engineer designs a [software tool] that can be fully defined in 20 pages whose rival product has been inadequately defined in 100 pages . . . ."

Perhaps the designers of software tools should stop designing "utilities" and start designing "efficiencies," "elegances," and "beauties."

Whether we call programming a craft, an art, an applied science, an engineering discipline, black magic, or a religion (you thought this editorial had nothing to do with Forth, didn't you?), we should not lose sight of the importance of its design element. Starting next month in our new column The Software Designer, we will call upon some working software designers to help us examine what makes good design, and why that question has different answers in different task domains. We'll look into different programming styles and ask why some programmers work in Forth or C while others use Ada or Modula-2. We'll ask designers what (currently nonexistent) tools they would most like to have. We'll examine software design in diverse environments: under Unix, on a Mac (no, not "for a Mac, on a Lisa"), and in logic (i.e., Prolog). And we'll examine the kind of programming this magazine was founded to foster back in 1976: highly-efficient code, or as we put it then, "running light without overbyte."

Michael Swaine

Michael Swaine

Introducing 3 New 68000 FORTH Systems

Multi-FORTH™ Under CP/M™-68K

- Available Now
- 32-bit Multitasking FORTH System
- Snapshot features allow creation of executable CP/M programs (in under 5 seconds from memory)
- Full access to CP/M files and facilities
- Inline Macro Assembler
- Line, Screen Editors, other utilities
- Introductory price.....**$695.00** (Regularly $895.00)

MacFORTH™

- Stand Alone Programming on Macintosh
- User defined MENUS
- User defined WINDOWS
- Compatible with Macintosh User Interface
- Mac File Compatability
- Interactive COMPILER & INTERPRETER
- User Access to TOOLBOX
- 32-bit FORTH System
- Large Installed Base with Local User Groups
- Available Now.....**$149.00**

Multi-FORTH™ Under UNIX™

The Speed and Flexibility of FORTH with the Structure and Utilities of UNIX™

- Comprehensive 32-bit FORTH Environment Under UNIX
- Full access to UNIX File Features and Shell from FORTH (ls, grep, rm, cat, etc.)
- Multi-FORTH Core Resident Real time Multitasker
- Inline Macro Assembler
- Ability to snapshot memory image as a loadable application
- Available in July on the Perkin Elmer 7300 (soon on other UNIPLUS Hosts)
- Color Graphics on the Perkin Elmer version
- Introductory price.....**$1295.00**

68000 FORTH Systems also available on HP Series 200 and Motorola VME10

For more information contact

CREATIVE SOLUTIONS

4701 Randolph Rd. Ste.12 Rockville, Maryland 20852
(301)984-0262

UNIX is a registered trademark of AT&T • CP/M is a registered trademark of Digital Research

Circle **no. 17** on reader service card.

Making Languages English Independent

Dear *DDJ*,
I was delighted to see the article on Chinese Forth in the June 1984 issue of *DDJ*. In many ways this is one of the most significant articles you have ever published because it recognizes for the first time that not everybody, or even a majority of the world's population, speaks English, and that to use a programming language based on English is to deny the non-English speaking majority easy access to the pleasures of personal computing.

There are two main approaches to rectifying the problem. One, as exemplified by APL, is to eschew natural language words altogether in favor of ideographic symbols. The second approach, used by Timothy Huang, is to take an existing language that allows redefinition of function names, and then rename them in the new language.

Probably the easiest computer language in which to do this is SAM76, described in your pages a few years back. It actually possesses a primitive function '@cn' that can be used to change the name of any primitive function. Using it, I have created a version of SAM76 that has all its primitives named in Polish. A new language called HERPES that I am currently working on also includes this feature.

It is a pity more was not said about the problems of printing Chinese text. For some natural languages, the problem of printing and displaying texts may easily exceed that of changing the keywords in the programming language. Chinese presents particular problems both in the complexity of the characters and the sheer quantity of them. Others with a lesser order of complexity are Arabic (with context-dependent letter shapes) and the Devanagari script (used for Sanskrit, Hindi and other tongues of North India) where two or three letters are regularly combined to form composite characters, even across the word boundary. I am currently struggling with the problem of printing Sanskrit on a Gemini 10X and it is not a trivial exercise.

Earlier I mentioned SAM76. Does anybody else use it? I haven't heard anything about it for about five years now and wondered if I were the only user. I find it unexcelled for programs manipulating character and text data, in fact most things except number crunching, where it is slow and suffers from awkward syntax, though the unlimited precision is often useful. Finally, for the linguists among you, I enclose a piece of text printed on a Gemini 10X using a polyglot word processing program written in SAM76 (see Figure 1, below). What language is it? (*No* prizes!)

Sincerely,
Greg Trice
1131 Sandhurst Circle #111
Scarborough Ontario
M1V 1V5, Canada

Dear *DDJ*:
I was very impressed with the ideas presented by Timothy Huang in his article "First Chinese Forth—a Double-Headed Approach" which appeared in *DDJ* No. 92 (June, 1984). Mr Huang has brought attention to a topic that merits further discussion and exploration. In my view, the problem of natural language barriers to the use of computer languages should be attacked at several levels. One solution is the use of translated keywords, as supported by Chinese Forth. For introductory teaching applications, instructions given to the computer should be as natural as possible. Thus, BASIC, Logo and other teaching languages should probably have translated command-sequences whereever English is sufficiently foreign to inhibit the use of the computer. However, software portability would become infeasible if each community utilized its own keywords to the exclusion of all others. This dilemma requires its own solution.

If computer language keywords were translated for each of the major natural languages, we would have a virtual tower of Babel in the computer world. At the moment, we have many different programming languages, but these languages are typically driven only with English-derived instructions/keywords. Programs created under Chinese Forth would not only be foreign to English speakers, but would also be foreign to Indonesian Forth, French Forth or even Japanese Forth programmers. Under such circumstances, programming communities would factionalize along natural language lines. Sharing software would be less possible, and the literature of any specific community would be much less valuable to other communities. This would magnify the problem of repetition of effort and would most seriously hurt nations that are underde-

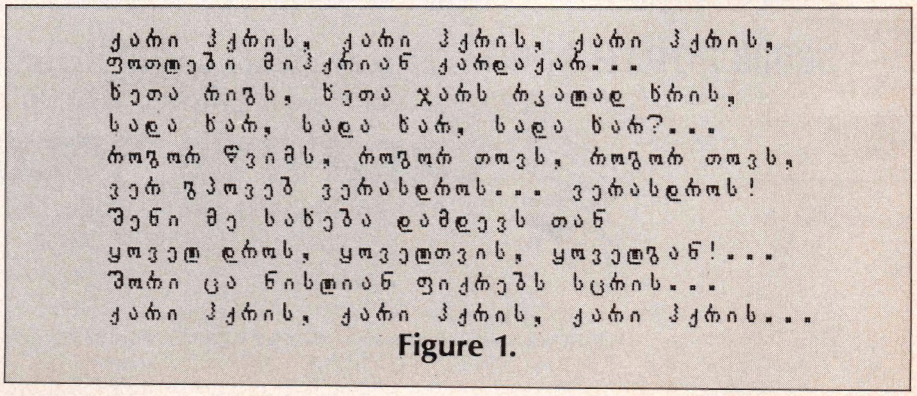


Figure 1.

veloped in the computer sense.

One solution to the problem might be to restrict programming to languages that lack any English keywords. This remarkably strong restriction essentially leaves only APL or direct, machine-code programming. APL is suitable for a large number of applications, but isn't for everyone. Furthermore, asking a whole world of programmers to work with one language is not reasonable in the least.

On the other hand, an important lesson can be gained from looking at APL's instruction mnemonics. These symbols suggest a compromise between keyword translation and the use of APL. We could create a set of symbolic keywords for a variety of languages; these keywords would bear no English dependence. Such keywords would ensure some degree of machine readable portability between communities, and simultaneously allow listings in the literature to become more accessible.

In order to demonstrate this idea, I devised a set of symbols for the C programming language (see Table 1, page 10). These symbols replace the compiler keywords and preprocessor commands as listed in *The C Programming Language*. Since C has no intrinsic functions, there was no need to create symbols for a large set of such internal functions. (Creating symbols for the standard input-output library names and definitions is, I believe, beyond the scope of this discussion).

The symbols presented in Table 1 themselves represent a compromise. They were selected from among those available on an IBM Personal Computer (or compatible). This was done so that this software concept could be used immediately without the need to invent new character generators, as with APL. However, these symbols are reasonably good and could be a practical symbolic replacement for C's English keywords. Rationale for the choice of the various symbols is included in Table 2 (page 10).

Many programmers will find the symbolic keyword alternative inferior to the use of natural language keywords. However, the symbolic language concept is still useful. It offers a means by which programmers can share software. Once they have the symbolic form of the program, a straightforward translator could be used to convert the code to their own particular keyword syntax. A reverse translator could be used before software is distributed (or published). Yet another translator could preprocess foreign keywords (into English equivalents) so that standard compilers could continue to be used.

The symbolic language does not help with documentation aspects of portability. Software documentation would have to be translated, as would command prompts and other text within the code itself. (I see no clear way to solve this problem, except by encouraging documentation in multiple natural languages.)

In this brief commentary, I've addressed another approach designed both to open the computer world to non-English speakers and to have an impact on keyword translation to software portability. I hope that others will feel free to comment in order to continue the dialog in this area.

Sincerely,

Anthony Skjellum
Pyramid Systems, Inc.
1695 Shenandoah Road
San Marino, CA 91108

DDJ

Symbolic Replacements for C Language elements

C Keywords

±	(int)	Γ	(char)	Φ	(float)	Θ	(double)	ϵ	(struct)
∩	(union)	‖	(long)	¢	(short)	╫	(unsigned)	α	(auto)
▵	(extern)	ϕ	(register)	f	(typedef)	¥	(static)	↑	(goto)
↓	(return)	╬	(sizeof)	↨	(break)	↕	(continue)	¿	(if)
♦	(else)	Σ	(for)	δ	(do)	Ω	(while)	≡	(switch)
→	(case)	←	(default)	♫	(entry)				

C Preprocessor controls

#¿	(#if)	#¿≡	(#ifdef)
#¿!≡	(#ifndef)	#♦	(#else)
#?	(#endif)	#£	(#line)
#≡	(#define)	#!≡	(#undef)
#¶	(#include)		

Miscellaneous

§( ) main( )

Table 1.

Symbol/IBM Chr. #	Keyword	Rationale
± 241	(int)	Symbol indicates a signed data type
Γ 226	(char)	Gamma is the third Greek letter, analogous to 'c' of char
Φ 232	(float)	No specific reason
Θ 233	(double)	No specific reason
ϵ 238	(struct)	Epsilon looks like mathematical 'belongs'
∩ 239	(union)	Intersection sign is opposite of union sign
‖ 186	(long)	Double bar through a negative sign
¢ 155	(short)	No specific reason
╫ 215	(unsigned)	Double bar through a negative sign
α 224	(auto)	Alpha is the first Greek letter, analogous to 'a' of auto
▵ 127	(extern)	Character points upward; external data normally at top
ϕ 237	(register)	No specific reason
f 159	(typedef)	Like an f sometimes used in mathematical definition
¥ 157	(static)	Symbol is similar to the symbol for electrical ground
↑ 24	(goto)	Upward arrow to symbolize a jump or goto
↓ 25	(return)	Downward arrow symbolizes going to end of block ie return
╬ 206	(sizeof)	Symbol looks like measurement device
↨ 23	(break)	Two way arrow with bottom supposed to indicate break
↕ 18	(continue)	Two way arrow less bottom indicates continuation
¿ 168	(if)	Upside down question-mark indicates interrogative
♦ 4	(else)	Diamond symbolizes decision-diamond from flowcharting
Σ 228	(for)	Mathematical summation symbol; often done with for loops
δ 235	(do)	Delta is fourth Greek letter, analogous to 'd' of do
Ω 234	(while)	No specific reason
≡ 15	(switch)	Symbol looks like a mechanical switch
→ 26	(case)	Right pointing arrow indicates item to follow
← 27	(default)	Left pointing arrow opposite of case ie default
♫ 14	(entry)	No specific reason

C Preprocessor controls

#if	#¿	Symbolizes interrogative
#ifdef	#¿≡	Reads 'if defined'
#ifndef	#¿!≡	Reads 'if not defined'
#else	#♦	Decision diamond symbolizes else
#endif	#?	Endif is opposite symbol to if
#line	#£	Pound sign supposed to look like a fancy 'l' for line
#define	#≡	Mathematical define symbol
#undef	#!≡	Reads 'un-define'
#include	#¶	Paragraph mark since this causes more text to be read

Miscellaneous

§( ) main( ) Section mark to indicate main program section

Table 2.

PRESENTING
The first compiler for dBASE II®

WordTech Systems is proud to announce the first compiler for dBASE II®. And we are introducing it with a special offer.

INDEPENDENCE

Now you can write compiled, efficient programs that will execute independently of dBASE II, and without RunTime®.

NO LICENSE FEES

You only buy dB Compiler™ once. You may compile as many applications as you wish, FOREVER, with no additional fees.

SPEED

Application programs are compiled into low level code and only include program functions that are absolutely necessary.

SECURITY

Compilation is far better than encryption for protecting your programming insights and procedures.

PORTABILITY

Using dB Compiler's cross-linkers you can use one development system to generate code for various target environments.

Suggested retail price: $750; additional target modules: $350
Special Offer: Compiler and an additional target module: $750
Offer expires 7/15/84. Corp./multi-user licenses available.

dB COMPILER™

WORDTECH SYSTEMS P.O. Box 1747, Orinda, CA 94563 (415) 254-0900

dBASE II, RunTime® Ashton-Tate

Circle no. 96 on reader service card.

DR. DOBB'S CLINIC

by D. E. Cortesi, Resident Intern

Lose That Ugly FAT

"Just as your mailbag ran dry," writes Chet Floyd, "the FAT on my IBM PC XT's hard disk was clobbered during a Pascal compile." Uh, sorry, Chet. And the rest of you take warning—keep those letters coming or make lots of backups! But to return to Chet's story . . .

"It was a disastrous encounter with what I suspect is a bug in PC-DOS 2.00. PAS1 used the first sector of the FAT as a directory. I do not have a definite answer to the problem, just a couple of theories you may wish to explore or explode.

"First, the players: IBM-DOS 2.0, MS Pascal 3.13, a nearly full hard disk, and 640K of memory busy with an AST memory disk and spooler, Btrieve, and MultiJob running 250K and 128K partitions. The compiler was in the larger partition, Personal Editor in the smaller. I've used this configuration for at least seven months, with 3.13 replacing the 1.0 compiler three months ago. The conglomeration has worked flawlessly.

"Then the crash occurred, with PAS1 vaguely indicating a 'File Access Error.' When I got around to inspecting the FAT, it was obvious that the first sector had been used for the directory entries of the work files PAS1 builds; there were three entries there, and the rest of the sector was set to zeros. Of course, the backup copy of the FAT was similarly bent.

"This caused the 300-odd clusters controlled by this FAT sector to 'vanish.' About 1.5 megabytes instantly became free; this didn't include the files controlled by directories that had resided in those 300 clusters.

"The theory I favor is that DOS cannot recover the multilevel error that results when both the current directory and the device are full. When PAS1 tried to create the first work file, DOS inspected the FAT and found no space, leaving the disk sitting at the first sector. But this error was perhaps not passed back, and DOS just used the current disk location to create another cluster for the directory. An alternative theory is that MS Pascal does its own disk I/O.

"Upon notifying Microsoft, I was told of a new version of MS Pascal, 3.2, which is supposed to have better disk-full recovery. I ordered it immediately, because *new version* is the code word for *your bug is fixed.* I asked why I, as a registered ISV and MS Pascal user, hadn't been notified that version 3.13 had been superseded. 'We don't have the database built yet, but we're trying to get to it.' So much for ISV support.

"I also contacted B&L Computing, the developer of MultiJob. To his credit, Bob Hudson, who wrote MultiJob, said he wished he could absolutely rule his code out of responsibility for the crash, but wouldn't, because of the black-box nature of parts of DOS. However, I am inclined to rule out MultiJob as the culprit.

"Although my backup procedures ensured that I would suffer no permanent loss of data, I was surprised at how much I had depended on my ability to reconstruct files as an excuse not to back them up. Reconstruction took several days and was traumatic enough to dissuade me from trying to force the condition or test my theories in any other way. I consider myself lucky to have been able to recover, because my backup strategy had never been subjected to the acid test of actual recovery."

Comments, anyone?

A RAM-Drive is not a Sheep Roundup

Reader Leonard Schwab was interested in our report on installing an Electra-Logics Quasi-Disk. He had just completed a more challenging installation. "Recently, I obtained a CompuPro M-Drive-H board for my six-year-old IMSAI VDP-80.

"CompuPro provides very little information about installing the M-Drive in a non-Godbout system. The documentation talks about installation software that did not come with my board. There is a model driver routine in the manual, but some crucial points are omitted. You must analyze the model driver to discover the nature of the data required by the board during accesses, and nothing is said about the technicalities of formatting the board on a cold start. Needless to say, this only sweetened the challenge."

Schwab eventually ended up rewriting all the disk-handling code of his Fischer-Freitas BIOS. In the process, he was able to add automatic density-sensing for his Persci floppies. He found that he could copy the system image to the M-drive during a cold start, so that warm starts became almost instantaneous, and he adjusted the system so that submit jobs would run with no disk accesses at all. These actions transformed his system, he says.

If anyone else is having trouble integrating the CompuPro M-drive into a non-CompuPro system, Schwab is willing to advise. Send us a note and we'll pass it on.

I'm OK, UART Confused

Yeah (sigh), wouldn't it be nice if CompuPro's manuals were as good as the boards they describe? We just spent 15 hours integrating an Interfacer 4 into our system. It replaces an antique Interfacer 1, one of whose two ports died when we misjumpered it.

Fifteen hours is too damn much time. Now, it isn't all CompuPro's fault. We spent the largest part of the time trying to comprehend the interfaces between the software and the board, between the board and the RS-232 connection, and between RS-232 and the devices that we wanted to con-

nect to the system.

But the manual could have been a lot more help. For instance, it could have used a consistent way of referring to the three serial ports on the board. If you have an Interfacer 4, memorize this list:

- User 1 or User 5 is Port C; it's on the right.
- User 2 or User 6 is Port B; it's in the middle.
- User 3 or User 7 is Port A; it's on the left.

Simple, isn't it? (No!) It would have been even nicer if the writer had used standard terminology for the RS-232 configurations: DCE and DTE, not "master" and "slave" (now, which was "master"? Oh yeah, the terminal—I mean, DTE . . .).

Because the serial ports are the same on at least three CompuPro boards (the System Support 1 and the Interfacers 3 and 4), we documented what we learned. We present it here for the benefit of any reader that needs it.

DCE and DTE

There are two actors in a standard RS-232 connection. They are called the data terminal equipment, or DTE, and the data communications equipment, or DCE. The names are standard-ese for "terminal" and "modem"; that is, a terminal-type device normally plays the DTE role, and a modem-type device plays the DCE role.

The RS-232 interface is used between many sorts of devices. Sometimes it isn't obvious what role a device should be expected to play. In general, a device that is an original source of (or a final sink for) data, will play the DTE role. A printer, a plotter, a digitizing tablet, or, of course, a terminal, will play the part of the DTE. The other actor in the connection—usually a computer or a modem—must take the role of DCE.

A personal computer may be called on to play either role, depending on the application. When the computer is connected to a device that is clearly a DTE, the computer must act as the DCE, but when the computer is hooked to a modem, it must act as the DTE. When two computers are connected directly to each other, it doesn't matter

which one plays which role—but both cannot play the same role.

One more point: the DCE actor normally is equipped with a female DB-25 socket, while the DTE actor is normally equipped with a male DB-25 plug. The presence of a female connector should indicate that a device is wired to play the DCE role (check your modem), and the presence of a male plug should indicate that the device is meant for use as a DTE.

This rule is followed most of the time with modems and terminals, but you can't depend on it when you are looking at the back of a computer.

Signal Lines

In most applications, there are only seven significant lines in the RS-232 interface (aside from ground and signal-ground lines). Here they are with their standard pin numbers in the DB-25 connector:

- Pin 2 carries data from DTE to DCE.
- Pin 3 carries data from DCE to DTE.
- Pin 4, Request to Send (RTS), is asserted by the DTE when it has data to transmit.
- Pin 5, Clear to Send (CTS), is asserted by the DCE when it is OK for the DTE to send.
- Pin 6, Data Set Ready (DSR), is asserted by the DCE when it is on-line and ready.
- Pin 8, Data Carrier Detect (DCD), is asserted by the DCE (a modem) when a connection is established over the telephone line.
- Pin 20, Data Terminal Ready (DTR), is asserted by the DTE when it is ready and on-line.

Let's repeat that list from the viewpoint of each device. As the DTE (the terminal, printer, and so on) sees the interface,

- It transmits on pin 2.
- It listens on pin 3.
- It asserts pin 4, RTS, when it wants to send.
- It awaits pin 5, CTS, before sending.
- It asserts pin 20, DTR, as long as it is ready for operation.

In addition, a true terminal may use pin 8, DCD, as a qualifier of its other operations. The Diablo 1650 KSR terminal, for example, lights a green "proceed" light when it detects DCD true. Many DTE devices will use pin 20, DTR, as a sort of reverse-CTS, dropping it when some temporary condition puts a halt to their operations. A printer, for example, may drop DTR when it is out of paper.

Now, from the standpoint of the DCE actor, we have these viewpoints:

- It listens on pin 2.
- It transmits on pin 3.
- It asserts pin 5, CTS, when it can take data from the DTE.
- It asserts pin 6, DSR, when it is on line and ready.

A true modem will assert pin 8, DCD, when it detects the data tone on the phone line. A half-duplex modem will take the DTE's signal on pin 4, RTS, as a sign that it should initiate a line turnaround in preparation for transmission and will not assert pin 5, CTS, until the turnaround is complete.

A personal computer playing the DCE role has no phone line to listen to and will often use these lines in a simpler way. It might, for example, simply

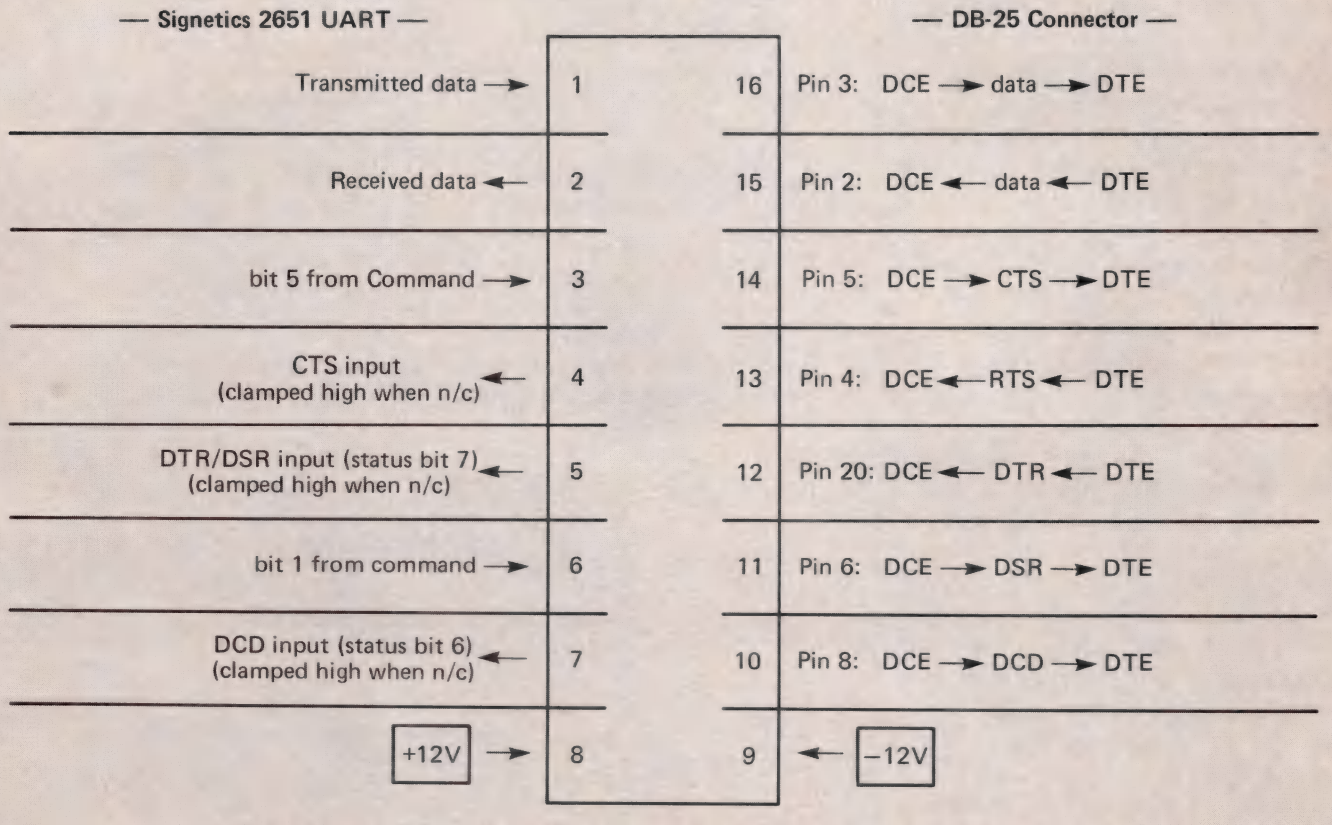


Figure 1 The DIP-socket interface between UART and DB-25.

short the DTE's pin 4, RTS, back out on pin 5, CTS, supplying an instant "Clear to Send" without effort.

The UART Connection

The 2651 UART is used on the CompuPro System Support 1, Interfacer 3, and Interfacer 4 boards. In each case, its connection to the RS-232 link is the same. There is a single, 16-pin, DIP socket for each UART. Seven lines are brought from the UART to pins 1 – 7 of the socket, while seven lines are brought from the DB-25 connector to pins 16 – 10, the facing positions on the socket. The arrangement is shown by the diagram in Figure 1 (page 14).

Let's run down the UART's signals as they appear on socket pins 1 – 7:

- Socket pin 1 carries data transmitted from the computer
- Socket pin 2 receives data from the connected device
- Socket pin 3 carries bit 5 from the command last written to the UART, translated to RS-232 voltage levels
- Socket pin 4 is presented to the UART as Clear to Send signal
- Socket pin 5 is passed through to become bit 7 of the UART status byte
- Socket pin 6 carries bit 1 from the command last written to the UART, at RS-232 levels
- Socket pin 7 is passed through to bit 6 of the UART status byte

Two of the socket pins (3 and 6) are driven directly by command bits that your software writes to the UART. These bits pass through an even number of level inversions on their way to the socket; hence, the socket levels reflect the bit values in the command. A one bit in the command will appear as a positive voltage at the socket; a zero bit will apppear as a negative voltage.

Two of the pins (5 and 7) are passed through to the UART status byte (and have no other apparent effect on UART operation). Like the command bits, these status bits reflect the voltages applied—a positive voltage will produce a one bit, a negative voltage appears as a zero bit.

The Clear to Send input on socket pin 4, the status input on pin 7, and (on the Interfacer 4 but not on the System Suppport 1) the status input on pin 5 are all pulled up to positive voltage lev-

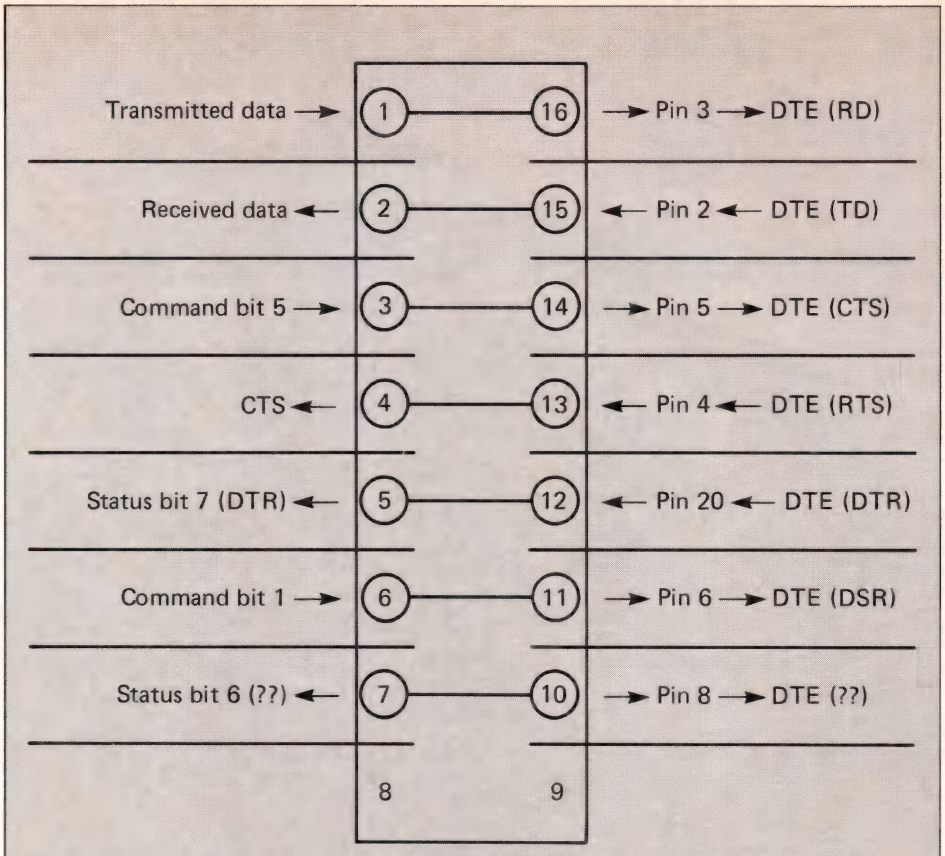


Figure 2 The effect of a DIP shunt is to make the UART play the DCE role.

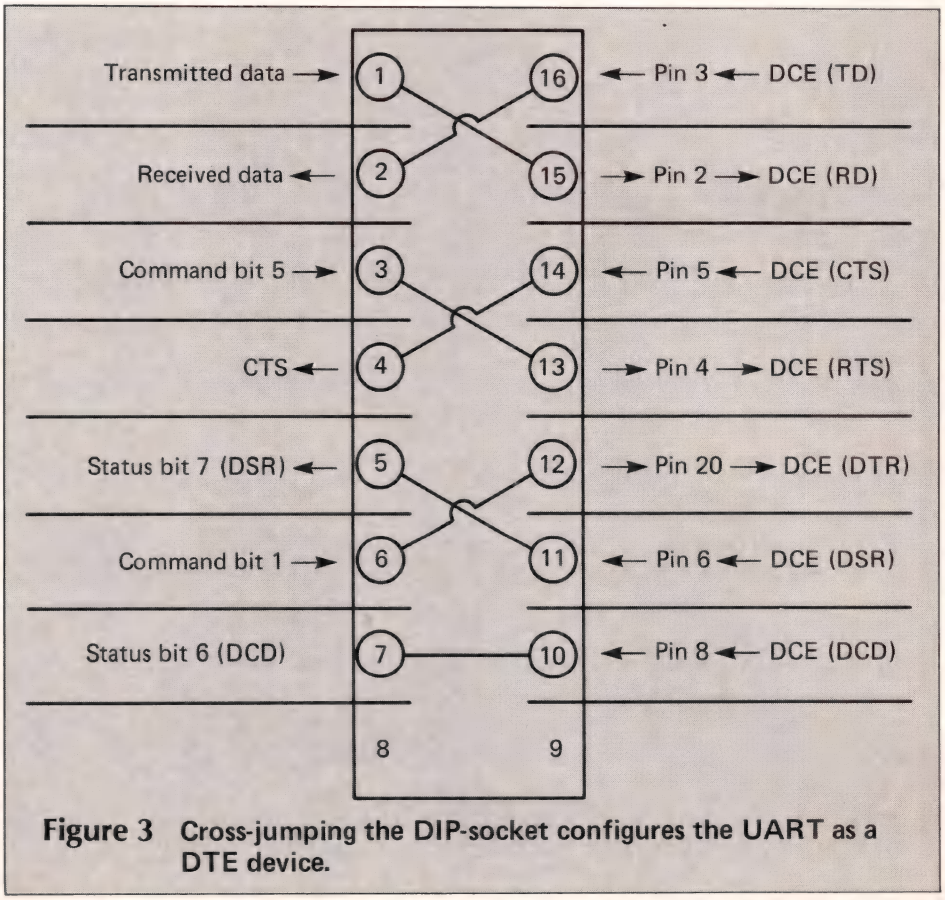


Figure 3 Cross-jumping the DIP-socket configures the UART as a DTE device.

els by a resistor. Thus, if these pins are not driven, they will appear positive.

Configuring the Port

You use this DIP socket to connect the UART to the DB-25 connector. In doing so, you configure that port as either a DCE or a DTE. This configuration is determined by the way that pins 1 – 7 are wired across to 16 – 10. Pins 8 and 9 on the socket are supplied with constant voltages for your convenience; you can use them as a means of tying a DB-25 pin to a fixed level. In a simple connection, you can use the socket as a means of connecting RTS back to CTS or for doing other tricky things. Here, however, we will examine the full configurations for the DCE and DTE roles.

The DCE Configuration

Examine Figure 2 (page 15). It shows the effect of installing a DIP shunt in the socket. Notice that the conductor between pins 8 and 9 *must be broken*; if you use a new, unmodified DIP shunt, you will be placing a direct short from +12 volts to −12 volts. Be careful about this; your CompuPro board may have been delivered with unmodified DIP shunts in these sockets.

The result of shunting each pin 1 – 7 to its facing pin 16 – 10, as shown in Figure 2, is to make this port a DCE. That is, the UART will be transmitting on pin 3, receiving on pin 2. You have direct control of RS-232 pins 5, CTS, and 6, DSR, through the UART command byte. Pin 8, DCD, will be held to a positive level by the pull-up resistor. The DTE's signal on pin 20, RTS, will be available in the UART status byte for your device driver to test.

There are a couple of anomalies in the configuration of Figure 2. The DTE's signal on pin 4, RTS, will be passed to the UART as "clear to send." It seems to cause no problem. And bit 6 of the status byte will always be one, since socket pin 7 is driven by neither device and hence will be pulled high. Nevertheless, this configuration works quite well as a DCE actor.

The DTE Configuration

Now turn to Figure 3 (page 15). This is how the DIP socket must be wired if the port is to be configured as a DTE. You must solder up this wiring using a DIP header. Furthermore, when configuring a port as a DTE you should bring it out on the back of the computer with a male plug. If you don't, you'll need an RS-232 "sexchanger" cable to connect it to the female socket on a standard DCE device. Alas, all the CompuPro-supplied interface cables terminate in female sockets at all positions, so you'll have to make up your own ribbon cable.

Be that as it may, let's consider the effect of the wiring shown in Figure 3 (page 15). The UART transmits on DB-25 pin 2 and receives on pin 3, as a DTE device ought to do. Your software can control the DTE signals of pin 4, RTS, and pin 20, DTR, through bits of the UART command byte. The attached DCE device's signals on pin 6 (DSR) and pin 8 (DCD) will be available in the UART status byte. The DCE's pin 5 signal, CTS, will be delivered to the UART as "Clear to Send."

UART Command and Status

Figure 4 (page 16) summarizes the status byte presented by the 2651

	D7	D6	D5	D4	D3	D2	D1	D0
DCE config.	DTR	n/a	Framing Error (or Break if Data=00h)	Overrun Error	Parity Error	Tx–Shift Reg. Empty – or – Change in DSR/DCD	Input Data Available	Output Clear
DTE config.	DSR	DCD						

Figure 4 The UART status register. Its meaning depends in part on the configuration of the DIP socket.

	D7	D6	D5	D4	D3	D2	D1	D0
DCE config.	0	0	CTS	Reset Error Flags in Status	Force Break	Enable Receiver	DSR	Enable Transmitter
DTE config.			RTS				DTR	

Figure 5 The UART command register. Its meanings depend in part on the configuration of the DIP socket.

UART under these configurations. Six of the bits reflect the UART's operations. Bit D7 reflects the voltage state of socket pin 5; that is, either pin 20, DTR, or pin 6, DSR, depending on the socket configuration. Either way, it means "other device ready."

Status bit D6 reflects a modem's pin 8 signal, DCD, when the socket is configured as a DTE. If the socket is configured as a DCE, this bit will always be high. Of course, you could modify the wiring of Figure 2 (page 15) to bring another signal to this bit.

Figure 5 (page 17) summarizes the use of the UART command byte. A command can be written to the UART at any time after it has been initialized; it isn't necessary to write the two mode bytes before every command. Four of the command bits control UART operations. Two of them are simply passed through to the configuration socket and from there to the DB-25 connector, however.

Command bit D5 controls the level at socket pin 3. When the port is configured as a DCE, this bit will be seen by the DTE as a CTS signal. In most applications it should be set to 1, because most DTE devices want a positive CTS (but some may not). When set up as a DTE, command bit D5 controls RS-232 pin 4, RTS. Most DCEs expect this signal to be positive and will turn it around as pin 5, CTS.

Command bit D1 controls the level at socket pin 6 and, hence, when the port is a DCE, it controls the level of pin 6, DSR. Most DTE devices expect this signal to be high when their modems are on-line. When the port is DTE, command bit D1 controls the RS-232 signal pin 20, DTR. Most modems expect this line to be positive when their terminals are on-line.

Thus, in most applications, both command bits should be written as 1 bits. This may not be true in all cases, however, because the world of RS-232 devices contains more exceptions than it does normal cases. At least, once you understand the requirements of the attached device, you can satisfy them through some appropriate wiring of the configuration socket and some setting of the command bits. **DDJ**

Reader Ballot
Vote for your favorite feature/article.
Circle Reader Service **No. 190.**

by Robert Blum

Of all the modules needed to form a complete banked version of CP/M Plus, the MOVE module is the smallest in size but perhaps the most difficult to debug. The three subroutines residing in the MOVE module form the nucleus of logic to use banked memory. The close relationship that exists between these routines and the hardware can be the cause of many mysterious system crashes, especially when you bring up CP/M Plus for the first time.

CP/M Plus: Interbank Memory Moves Without DMA

I was at first disappointed after reading the CP/M Plus documentation because it insists that DMA hardware be used for interbank data movements. My machine has a DMA disk controller, but no way of asking it to perform a memory-to-memory move operation. After thoroughly studying the sample BIOS listings included in the system manual, however, I couldn't think of any reason to doubt that I would be able to find a programmable solution to this problem.

The listing on page 20 shows the MOVE module that I am currently using. It is patterned after Digital Research International's (DRI) suggested routine with only a few small modifications. So far it has proven very reliable and with only two changes should work equally as well in most any Z80 hardware configuration with or without special DMA features.

The first routine, XMOVE, is provided for banked systems that support memory-to-memory transfers over the entire extended address range. Systems with this feature can have their data buffers located in an alternate bank of memory instead of in common memory. By relocating the data buffers away from common memory (application program) you easily can achieve a TPA of 60K or larger.

A call to XMOVE affects only the following MOVE call. Each time you make an interbank transfer, you must first call XMOVE to establish the data source and destination bank numbers before calling MOVE.

The second routine, MOVE, performs the memory-to-memory block move operation. My version is slightly enhanced to allow any size interbank move. The sample routine only permitted 128 bytes to be transferred at a time, which for me would cause excessive overhead. My hardware will not support direct interbank memory moves so I used a combination of Z80 block move instructions to perform the move in segments of 128 bytes each. This required that each memory segment be stored first in a common memory buffer and then moved to the correct destination address after the appropriate memory bank had been switched into context. This entire process is not nearly as complicated as it may sound, and outside of a need for special hardware, it is the most efficient way to implement interbank moves.

The last routine, BANK, is as dangerous as any in CP/M Plus. At entry to this routine the A register contains the number of the CP/M Plus memory bank (0 through 15) to be switched into context. Because I use the older style bank select method for memory bank switching, I first used a translation table to convert the CP/M bank number to one understood by my hardware.

As I said, the MOVE module is not large in size but does perform some of the most important functions in a banked version of CP/M Plus.

Z80ASM: An Assembler Language Development System

Continuing with July's discussion on Z80ASM from SLR Systems (an assembler language development system), the package I received contained a single 8-inch disk and a standard-size, three-ring documentation binder. There were six files on the disk: four executable programs, a sample assembler language source file for testing the package after installation, and a "READ.ME" file containing recent additions or updates to the documentation manual. I refer to Z80ASM as a package because a linkage editor and a librarian are also included. For the purpose of this discussion, I will restrict my comments to the assembler and the configuration utility program.

To prepare Z80ASM for use, you must run the system configuration utility program. During this setup procedure, you must answer a number of questions about the version of CP/M in use and how the assembler's many default parameters are to be set. Some of the questions are detailed enough that it would be wise to first study the configuration section of the manual before you attempt to run the configuration utility.

Many of the default parameters pertain to the format of the printed hardcopy listing. In addition to specifying how many columns across and lines down are to be printed on each page, you can select options to alter the page appearance. One option, for example, allows the line number assigned by the assembler to be printed on the left margin or between the listing of the generated object code and the actual source statement.

Another option that is particularly important to me is the inclusion of the date and time into the page headings. To gather the date and time from the clock, Z80ASM senses the operating system version in use and issues the appropriate BIOS function calls. If the CP/M system in use doesn't automatically support built-in clock functions, you can specify the memory address of a subroutine that can be called to pass along the clock data.

Z80ASM will also initialize the printer with up to eight characters of

user-specified control information before printing starts and again at the listings termination—certainly a very important feature for the narrow carriage printer owner.

The default filename extension for all input, output, and work files can also be configured to meet your standards. You can eliminate the monotonous task of keying the parameters that control the assembly options by specifying them once at configuration time (and then give them no further thought unless there is some special need). Of course, if you need a special option, it is not necessary to rerun the configuration utility. Practically every permanent option can be overridden if you specify an execution time parameter.

Running Z80ASM for the first time is where the fun begins. As I mentioned in the first installment of this review, I was skeptical when told I could expect a sixfold speed improvement. My first assembly, however, veraccurate and that my fears were unfounded. I'm told by the author that this incredible assembly speed is due largely to making every use of the Z-80's instruction set and to buffering all disk data into 1K segments. The data rates are even further enhanced when you can use the multi-sector function of CP/M Plus.

A full complement of operator commands are available to control the execution of Z80ASM. If you have ever sat blindly staring at your CRT questioning why a program is running so long, or worse yet, whether it is running at all, you will appreciate these features. If at any time it is necessary to prematurely end an assembly, simply typing a CTL-C will abort the job without your needing to resort to more drastic measures such as hitting the reset button. Further, assembly operation may be temporarily suspended if you key CTL-S and later resume by typing CTL-Q.

Runtime control of the console and printer devices is also provided. The console driver may be enabled and disabled (toggled) with a CTL-Z, and the list device driver may be toggled with a CTL-P. Note that both of these commands may be used whether or not their drivers were enabled in the command line. These useful commands let you view just a portion of the assembly without requiring you to insert control statements into the program source code. And in the unlikely event that you may be questioning what Z80ASM is doing at any moment, typing a question mark will display the current filename, line number, and source line that is being assembled.

Errors that creep into your assembly are described by over 30 full text messages. Not only are they printed in the listing, but they are also displayed on the CRT as the assembly runs.

To ensure that no error messages slip by, you will be asked during the configuration process for the number of lines to be displayed on the CRT before the assembler stops and waits for you to command it to continue. A maximum error limit can also be set that, when reached, automatically aborts the assembly. In any event, if the assembly ends with errors present, the CP/M Plus error code in the system control block is set to FF00.

A great deal of an assembler's usefulness comes from the flexibility of the pseudo-ops it recognizes as well as a well-integrated macro language. Again, Z80ASM goes one step beyond what is expected of a good assembler by providing several unique pseudo-ops that allow operator entry of values and the display of text on the CRT at runtime. Of particular interest to me were the string comparison pseudo-ops (all of this in addition to those carried forward from M80 and DRI's RMAC assembler).

In summary, Z80ASM is an extraordinary product that must be considered by anybody who does assembly language programming. Not only does the assembler work as advertised, it is also well supported.

If you want to know more about Z80ASM or the other products offered by SLR Systems, you can contact the company directly at 200 Homewood Drive, Butler, PA. 16001; in Pennsylvania, call (412) 282-0864, other states (800) 833-3061.

DDJ

(Listing begins on page 20)

Reader Ballot
Vote for your favorite feature/article.
Circle Reader Service **No. 191.**

CP/M Exchange Listing (Text begins on page 18)

```
                .z80
                title   MOVE - bank select & move module for CP/M Plus
                subttl  save areas and equates
                .comment||
                page

        ;
        ;       Global definitions for CP/M Plus
        ;

        false           equ     0
        true            equ     not false
        banked          equ     true
        ;
        ;


                maclib  cpm3glbl.lib

                cseg

        ;
        ;       subroutines for others modules use
        ;

                public ?move,?xmove,?bank
                extrn   ?pmsg,?pdec
        ;
        ;       external areas and subroutines that we may need to use
        ;

                extrn   @cbnk,@dskbf
                extrn   ?bnksl

        ;
        ;       local save areas
        ;

move_stat:      defb    0                       ; flag to show next move is interbank

src_des_bank    equ     $
src_bank:       defb    0                       ; source bank number
des_bank:       defb    0                       ; destination bank number

des_add:        defs    2                       ; destination address for interbank move
src_add:        defs    2                       ; source address for interbank move
mov_cnt:        defs    2                       ; byte count for interbank move

hold_hl:        defw    0                       ; saveareas for callers registers
hold_de:        defw    0
hold_bc:        defw    0

old_stk_save:   defs    2                       ; place to save users stack
                defs    16                      ; place for our local stack
local_stack     equ     $

tc_bank:        defb    0

conv_bank_tbl:  defb    01h,02h,04h,08h,010h,020h,040h,080h
                                                ; conversion table used to translate
                                                ; CP/M's bank to ours

        ;
        ;       local equates
        ;

bank_port       equ     040h                    ; bank selection port address
int_bnk_act     equ     0                       ; flag for interbank move active
```

(Continued on page 22)

THE PROGRAMMER'S SHOP™

helps compare, evaluate, find products. Straight answers for serious programmers.

SERVICES

- **Programmer's Referral List**
- **Compare Products**
- **Help find a Publisher**
- **Evaluation Literature free**
- **BULLETIN BOARD - 7 PM to 7 AM 617-826-4086**
- **Dealer's Inquire**
- **Newsletter**
- **Rush Order**
- **Over 300 products**

PUBLIC DOMAIN Research - Free

6 months paid research gives you leverage, learning. We found, combined, added to the best. All run, have source in C or ASM. Order $150 + get one free: Database, Editors, Modems, MSDOS RAMdisks & utils, Games in C.

RECENT DISCOVERIES

GRAPHIC C comes with full source for PC/MSDOS graphics for screens, printers, plotters. Particularly strong for scientific plots. Optional use of 8087. Up to 4096 x 4096 Res. Desmet, C86. $195

"C" LANGUAGE

	LIST PRICE	OUR PRICE
MSDOS: C86-8087, reliable	$395	call
Desmet with debugger	159	145
Lattice 2.1 - improved	500	call
Microsoft C 2.x	500	349
Williams - NEW, debugger	500	call
CPM80: Aztec by Manx	199	call
BDS C - solid value	150	125
ECOsoft C - now solid	250	255
APPII: Aztec - full, decent	199	call
MACINTOSH: First - by 7/15 ...	NA	385
Compare, evaluate, consider other Cs		

BASIC

	ENVIRONMENT		
BASCOM-86 - MicroSoft	8086	395	279
BASIC Dev't System	PCDOS	79	72
BASICA Compiler - BetterBASIC - 640K	PCDOS	—	325
CB-86 - DRI	CPM86	600	439
Prof. BASIC Compiler	PCDOS	345	325
MACINTOSH COMPILER with BASICA syntax	MPC	NA	325

Ask about ISAM, other addons for BASIC

FEATURES

C HELPER has source in C for MSDOS, CPM80 for: DIFF, GREP, Flowcharter, C Beautifier, others. Manage source easier. $125.

PROLOG86 Interpreter for MSDOS includes tutorials, reference and good examples. Learn in first few hours. For Prototyping, Natural Language or AI. $125.

EDITORS Programming

C Screen with source	8080/86	NA	75
FINAL WORD - for manuals	8080/86	300	215
MINCE - like EMACS	CPM, PCDOS	175	149
PMATE - powerful	CPM	195	175
	8086	225	195
VEDIT - full, liked	CPM, PCDOS	150	119
	8086	200	159

UNIX PC

COHERENT - for "C" users	PClike	$500	475
VENIX - "true V7" w/FTN	PClike	800	775
XENIX - "true S3" - rich	PC	1350	1285

Ask about run-times, applications, DOS compatibility, other alternatives UNIX is a trademark of Bell Labs

LANGUAGE LIBRARIES

C to dBASE interface	8080/85	$150	$140
C Tools 1 - String, Screen	PCDOS	NA	115
C Tools 2 - OS Interface	PCDOS	NA	92
GRAPHICS: GSX - 80	CPM80	NA	75
HALO - fast, full	PCDOS	165	165
Greenleaf for C - full	PCDOS	NA	165
ISAM: C Index + -no royalties	MSDOS	NA	400
BTRIEVE - many languages	PCDOS	245	215
PHACT - with C	PCDOS	NA	250
PASCAL TOOLS - Blaise	PCDOS	NA	115
SCREEN: PANEL-86 - many languages	PCDOS	350	315
WINDOWS for C	PCDOS	NA	139

Ask about many others for FTN, BASIC, PASCAL, C-ISAM, Screen, Stat, Graphics.

PASCAL

	ENVIRONMENT	LIST PRICE	OUR PRICE
PASCAL MT + 86	CPM86/IBM	$400	$279
MS PASCAL 86	MSDOS	300	215
PASCAL 64 - nearly full	COM 64	99	89

OTHER PRODUCTS

AKA ALIAS - improve DOS	PCDOS	NA	60
Assembler & Tools - DRI	8086	200	159
CODESMITH-86 - debug	PCDOS	149	139
Disk Mechanic - rebuild	MSDOS	70	65
IQ LISP - full 1000K RAM	PCDOS	175	call
MBP Cobol-86 - fast	8086	750	695
MicroPROLOG	PCDOS	NA	265
Microshell improve CPM	8080	150	125
Microsoft MASM-86	MSDOS	100	85
MS Fortran - improvements	MSDOS	350	255
PL/1-86	8086	750	495
PLINK-86 - overlays	8086	350	315
Polylibrarian - thorough	MSDOS	99	89
PROFILER - flexible	MSDOS	NA	175
Programmers Tlkt w/source	8086	NA	135
READ CPM86 from PCDOS	PCDOS	NA	55
READ PCDOS on an IBM PC	CPM86	NA	55
TRACE86 debugger ASM	MSDOS	125	115

Note: All prices subject to change without notice. Mention this ad. Some prices are specials.

Ask about COD and POs.
All formats available.

Call for a catalog, literature, and answers

800-421-8006

THE PROGRAMMER'S SHOP™
128-D Rockland Street, Hanover, MA 02339
Visa Mass: 800-442-8070 or 617-826-7531 MasterCard

Circle **no. 63** on reader service card.

PROLOG-86™

Learn Fast, Experiment

1 or 2 pages of PROLOG would require 10 or 15 pages in "C."

Be familiar in **one evening.** In a **few days** enhance artificial intelligence programs included like:

- **an Expert System**
- **Natural Language** **(generates dBASE display)**

Intro Price: $125 for PCDOS, CPM-86.
Full Refund if not satisfied.

CONTEST: "Artificial Intelligence Concepts"

$1,000 Prize, Recognition for applications in PROLOG-86™ that teach, are clear, illustrate. Call for details. Deadline 11/31/84

SOLUTION SYSTEMS™
45-D Accord Park, Norwell, MA 02061
617-871-5435

Circle **no. 81** on reader service card.

C HELPER™

UNIX™-like Utilities for C Programming with source

Save time when working with your C programs. Full source lets you make them work your way, helps you learn.

Utilities included: compare files (DIFF), cross reference your variables (CCREF), examine the flow of functions as they call each other (FCHART), format and indent programs (pretty printer), search for patterns (GREP). Others check program syntax, print programs your way and more.

UNIX is a trademark of Bell Labs.

$135 for PCDOS, MSDOS, CPM-86 or CPM80.

Call with questions or for "Programming with C Helper".

SOLUTION SYSTEMS™
45-D Accord Park, Norwell, MA 02061
617-871-5435

Circle **no. 82** on reader service card.

```
                subttl  ?XMOVE - interbank move setup subroutine
                page
        ;
        ; set interbank move flag active for next call to ?move subroutine
        ; and save both the destination and source bank numbers
        ;
        ; at entry:
        ;
        ; b = destination bank
        ; c = source bank
        ;
?xmove:
                ld      (src_des_bank),bc       ; save source and destination
                ld      a,(move_stat)           ; get status byte
                set     int_bnk_act,a           ; set interbank move active
                ld      (move_stat),a           ; save status byte
                ret                             ; return to caller

                subttl  ?MOVE - memory to memory move subroutine
                page
?move:
                ld      a,(move_stat)           ; test flag for interbank move
                bit     int_bnk_act,a           ; and go to appropriate routine
                jr      nz,int_bnk_mov          ; *

                ex      de,hl                   ; we are passed source in de and dest in h
                ldir                            ; use z80 block move instruction
                ex      de,hl                   ; need next addresses in same regs
                ret                             ; return to caller

        ;
        ;       this routine is improved over the DR example
        ;
        ;       interbank moves of greater than 128 bytes can be made with one exception,
        ;       a move cannot span a bank boundary
int_bnk_mov:
                res     int_bnk_act,a           ; set off interbank move flag
                ld      (move_stat),a           ; save status byte

                ld      (old_stk_save),sp       ; save callers stack
                ld      sp,local_stack          ; and load it with a local stack

                ld      (des_add),hl            ; save destination address
                ld      (src_add),de            ; save source address
                ld      (mov_cnt),bc            ; save length of move
int_bnk_mov_a:
                ld      hl,(mov_cnt)            ; restore move length
                ld      a,l                     ; check for zero length move
                or      h                       ; *
                jr      z,int_bnk_mov_ret       ; *
                ld      bc,128                  ; segment length
                xor     a                       ; clear carry flag
                sbc     hl,bc                   ; decrement length
                jr      c,int_bnk_mov_last      ; last move this time
                ld      (mov_cnt),hl            ; save remaining length
                ld      hl,(src_add)            ; restore source address
                ld      de,@dskbf               ; point at intermediate save area
                ld      bc,128                  ; segment length
                ld      a,(src_bank)            ; get source bank number
                call    ?bank                   ; find physical bank and swap it
                ldir                            ; move a segment
                ld      (src add),hl            ; save new source address
```

```
                ld      hl,@dskbf               ; point at segment as source
                ld      de,(des_add)            ; restore real destination address
                ld      bc,128                  ; segment length
                ld      a,(des_bank)            ; get segment bank number
                call    ?bank                   ; find physical bank and swap it
                ldir                            ; move a segment
                ld      (des_add),de            ; save new destination address
                jr      int_bnk_mov_a           ; loop until complete

int_bnk_mov_last:

                ld      hl,(src_add)            ; restore source address
                ld      de,@dskbf               ; point at intermediate save area
                ld      bc,(mov_cnt)            ; remaining length to move
                ld      a,(src_bank)            ; get source bank number
                call    ?bank                   ; find physical bank and swap it
                ldir                            ; move a segment
                ld      (src_add),hl            ; save new source address
                ld      hl,@dskbf               ; point at segment as source
                ld      de,(des_add)            ; restore real destination address
                ld      bc,(mov_cnt)            ; remaining length to move
                ld      a,(des_bank)            ; get destination bank number
                call    ?bank                   ; find physical bank and swap it
                ldir                            ; move a segment
                ld      (des_add),de            ; restore ending source address

int_bnk_mov_ret:

                ld      a,(@cbnk)               ; load current bank number
                call    ?bnksl                  ; find physical bank and swap it
                ld      sp,(old_stk_save)       ; restore callers stack pointer
                ret                             ; back to caller

                subttl  ?BANK - select bank subroutine
                page

?bank:

                ld      (hold_hl),hl            ; save callers registers
                ld      (hold_de),de            ; *
                ld      (hold_bc),bc            ; *
                ld      c,a                     ; desired bank number to bc
                ld      b,0                     ; *
                ld      hl,conv_bank_tbl        ; pointer to bank conversion table
                add     hl,bc                   ; create memory pointer to desired byte
                ld      a,(hl)                  ; load bank number
                out     (bank_port),a           ; swap in desired bank
                ld      a,c                     ; restore bank number
                ld      hl,(hold_hl)            ; restore user registers
                ld      de,(hold_de)            ; *
                ld      bc,(hold_bc)            ; *
                ret

                end
```

End Listing

File Maintenance in Forth

by Ray Cadmus

Last year I was asked by a group of ceramics shop owners to develop a computer system to help them run their businesses more efficiently. Since my wife, a shop owner herself, was the spokeswoman for the group, how could I say no? Their wants were few. Keep it cheap, simple and comprehensive—in that order! To keep it cheap I looked to the Commodore 64 computer where a complete system with disk and printer was available for less than $1000 (not the best choice, as it turned out). The need for simplicity dictated that everything must be self-prompting or menu-driven. As for comprehensive—the want list grew . . . .

The group needed the computer to function as an electronic cash register, capturing both financial and inventory control information. Automatic price, product description and customer information lookup would be "nice." In general, they wanted inventory control, mailing list maintainance, class scheduling and full accounting functions including vendor and customer histories. With so much data to handle and so many files to be built and maintained, the small computer had to fill a tall order. This article presents my response to the challenge a simplified means; written in Forth, that builds and performs much of the routine maintainance on many types of data files.

Forth's extensibility makes it ideal for providing a core on which to build more complex structures. This data-management core begs to be extended, and the author shows some data-management tools that can be built from it.

Ray Cadmus, 600 W. Lee, Moberly, MO 65270.

Why Forth?

When I set out to do a simple business application on a Commodore 64, I quickly discovered that BASIC on the C64 is just too limited to really do justice to the power and capabilities of the machine. Forth was readily available; it is fast enough for dynamic graphics and sound routines (although neither is demonstrated in this project); and it is extensible, so it can grow more naturally to handle the features of whatever computer uses it—besides, I had been looking for an excuse to learn Forth anyway.

File Maintenance and Forth

One of the more common functions of a small computer is the storing and retrieving of data. Whether you call this data management or file maintenance, it normally means writing a series of programs to build and maintain individual data files. The Forth screens in the listing on (page 26) present a series of routines or "words" that allow the very simple description of a data file and its associated screen format. The routines provide capabilities for adding new records and for changing existing ones. You may change or retain data within a record on a field-by-field basis. Once you have established the file maintenance routines as a part of the Forth system, file handling is a matter merely of your defining a table word, setting a starting BLOCK and re-

cord length, then stepping into the maintenance MENU for the rest of the operations.

Before we look at the details of the program, it may be useful for me to describe a few of the terms I'll be using and to examine how Forth's handling of data differs slightly from most conventional programming systems.

For most people, the smallest practical unit of data is the character or "byte" of data. We combine these characters into logical, meaningful groups to make up a data element or "field" of data. This field may be numeric, as in a quantity or price field, or it may be alphabetic as in a description field. We can combine related fields into a "record." For example—an ITEM RECORD may contain an ITEM NUMBER field, a DESCRIPTION field, a PRICE field and a QUANTITY-ON-HAND field. This grouping makes up a "logical record." A "file" is a collection of logical records stored on a device outside the computer—usually a disk drive.

Both records and fields may be either variable-length or fixed length. Variable length units are usually separated by a common delimiter of some sort such as a comma, but fixed-length units are not; for example, our item record:

123,BIG BOX,19.95,15 (variable)
00123BIG BOX 019950015 (fixed)

A variable-length item takes up less room but a fixed-length record is more predictable. For instance, because we know how long each record is, we can calculate where any record will be by multiplying the number of preceding records by the record length. This is the basis for random access files; that is, we can go directly to the desired record without searching through each record from the beginning of a file.

How is Forth different? Forth organizes and stores its data as 1024 character "physical records" known as BLOCKS or SCREENS, stored by block number. We may store our "logical records" into a Forth BLOCK.

The Forth package used originally in the development of this package was the Commodore 64 Forth by Performance Micro Products (PMP). I have since converted the file management system to run under Laboratory Microsystems (LM) Z80 Forth. The ease of this conversion is a testimony to the portability of Forth. PMP Forth and LM Forth both are based on FIG Forth, with a few extensions and a few differences. The major difference (and the reason that I used the LM version for the enclosed examples) is that PMP Forth uses a 24 × 40 screen format rather than the standard 16 × 64 format of LM Forth. I encountered several minor differences: NUMBER in LM Forth requires a trailing blank to terminate while PMP Forth accepts either a blank or a NUL; and PMP Forth uses ?DUP where LM Forth uses –DUP.

Screen Descriptions

Screen 105 describes the terminal control as used by my Radio Shack TRS-80 Model II, and screen 106 defines a simple, relative record file access system. The word RECORD takes the record number off the stack, reads the data block if necessary, then sets up a pointer to the record on the block. You can change RECORD to use the native operating system's file system instead of the Forth block structure if you desire. At the very minimum, you should change it to check for a maximum block value so that your file doesn't overwrite something it shouldn't. You may use record 0 to keep miscellaneous data about the file. Currently, the first two bytes in record 0 contain the number of records in the file. You should set this count to 0 for a new file. For example:

```
(0 BBLOCK @ block !)
```

Screens 107 and 108 contain some of the words used by the data entry portion of the system. ?DIGIT returns a true or false flag showing when the top stack entry is an ASCII digit. ?NUMBER determines when the text string pointed to by the top stack entry is convertible to a number. Using these words ensures that the program won't "blow up" from unusable garbage. ASK prompts for a yes/no and returns a true flag when the response is Y. LEN returns the length of a string. WAIT is a pause word used to keep data on the screen until you have viewed it. INPUT accepts a number from the terminal, converts it to internal form, and leaves it on the stack as a double number.

Screens 109 – 118 are the file maintenance system (FMS) words, and screens 119 and 120 are a couple of sample applications using the FMS. The key to the FMS is the word PBUILD, which builds a page description table. It is called with the top of stack containing the number of fields to be displayed on the data entry screen. Following the word PBUILD are the field description entries themselves (see screens 119 and 120 for examples).

The set of entries for each line of the PBUILD table are:

(1) The line on the screen to display the prompt
(2) The screen column for the prompt display
(3) The line on the screen to display entered data
(4) The screen column for data entry
(5) The length of data to enter
(6) The data type code where:
    0 – no data entry, display only
    1 – numeric single byte
    2 – numeric single word
    3 – numeric single word; treat as a dollar amount, display as nnnn.nn (2 decimals)
    4 – numeric double word
    5 – numeric double word; treat as a dollar amount
    9 – alphanumeric text
(7) The location in the record of the data (base 0)
(8) The prompt text delimited by trailing quotes

That's it! This simple table definition sets up everything for data entry, display, and update. Note that the type code determines how much room a numeric field takes up in the data record.

Screen 111 contains the FFIND routine that finds the location of the field description data in the table. FDISPLAY displays the prompt for a given field, and PDISPLAY displays the entire screen of prompts.

Screen 112 describes FDATA and ?FDATA, which determine where the data is in the record and what type it is. ?FDATA is used only by FDATA.

FGET in screen 113 uses FTEXT and FNUM to accept input data and place it into the proper record location. FPUT on screen 114 takes a field's data from the record and places it on the screen. PPUT and PGET on screen 115 force the display and entry, respectively, of a whole screen and record full of data.

The words in screens 109 – 115,

which form the guts of file maintenance, could also be used as screen and record managers in other programs. The rest of the words on screens 116 – 118 use the file and screen words to develop a simple FMS. You could easily enhance these routines to add record indexing and report generation for a full-fledged data base manager.

Screens 119 and 120 are examples of two applications using the simple FMS. The first is a standard name and address file system. The second maintains a mold catalog for a ceramics shop.

Summary

The steps required to build and use a file with this package are:

(1) Draw up the desired screen layout, and mark row and column positions for each field.

(2) Define the description table using PBUILD.

(3) Pick a range of blocks for the file, and set the record count in the first block to 0.

(4) Define the top-level application word to:

- Call IREC to initialize the starting block and record length
- Set up the page table pointer with PINIT
- BEGIN PMENU UNTIL

(5) Start the application by invoking the top-level word.

Once started, the maintenance function is menu-driven. Selecting the Add function adds a new record as the next sequential record number. You should modify List for the application; as it stands, it displays a record number and the first 20 characters of each record. Change and Display both select by record number. You could easily change Display to do a search on a specific field. Change displays the selected record as it currently exists and allows changes on a field-by-field basis. To step past a field without changing it, press Return. To change a field, type over it.

I hope this little system and its supporting routines prove useful. Keep in mind that this package, while useful as it stands, is really just the demonstration of a concept and is begging for expansion. Items that should be added are a record delete function, keyword indexing, and a full screen editor to assist in building the PBUILD table. In the meantime, it has served its primary purpose of helping me learn a little bit about Forth.

A Postscript

I commented earlier that perhaps the Commodore 64 was not the best choice for this project. I don't want to leave you with the wrong impression. The C64 is a great little computer for the money; its weakness for this business application is the slow speed of disk drives. Disk copies take so long to create that they would probably be ignored by the casual or inexperienced user; a lack of backup files might eventually prove disastrous.

DDJ

File Maintainance Listing (Text begins on page 24)

```
Screen # 105
  0 ( SCREEN CONTROL - XY & CLS FOR RADIO SHACK II P&T CP/M)
  1
  2 : XY     ( ROW COL ---    )
  3          SWAP 27 EMIT 89 EMIT
  4          0 MAX 23 MIN 32 + EMIT 0 MAX 79 MIN 32 + EMIT ;
  5
  6 : CLS    ( HOME AND CLEAR SCREEN)
  7          12 EMIT ;
  8
  9 -->
 10
 11
 12
 13
 14
 15

Screen # 106
  0 ( RECORD ACCESS )
  1
  2 0 VARIABLE RLOC          0 VARIABLE RLEN          0 VARIABLE RNO
  3 0 VARIABLE REC/BLK       0 VARIABLE BBLOCK
  4
  5 ( BBLOCK IS STARTING BLOCK FOR THE FILE )
  6 ( RLEN IS LOGICAL RECORD LENGTH )
  7
  8 : IREC   ( BBLOCK RLEN ---       INITIALIZE RECORD POINTERS)
```

(Continued on page 28)

COHERENT™ IS SUPERIOR TO UNIX* AND IT'S AVAILABLE TODAY ON THE IBM PC.

Mark Williams Company hasn't just taken a mini-computer operating system, like UNIX, and ported it to the PC. We wrote COHERENT ourselves. We were able to bring UNIX capability to the PC with the PC in mind, making it the most efficient personal computer work station available at an unbelievable price.

For the first time you get a multi-user, multitasking operating system on your IBM PC. Because COHERENT is UNIX-compatible, UNIX software will run on the PC under COHERENT.

The software system includes a C-compiler and over 100 utilities, all for $500. Similar environments cost thousands more.

COHERENT on the IBM PC requires a hard disk and 256K memory. It's available on the IBM XT, and Tecmar, Davong and Corvus hard disks.

Available now. For additional information, call or write,

Mark Williams Company
1430 West Wrightwood, Chicago, Illinois 60614
312/472-6659

COHERENT is a trade mark of Mark Williams Company.
*UNIX is a trade mark of Bell Laboratories.

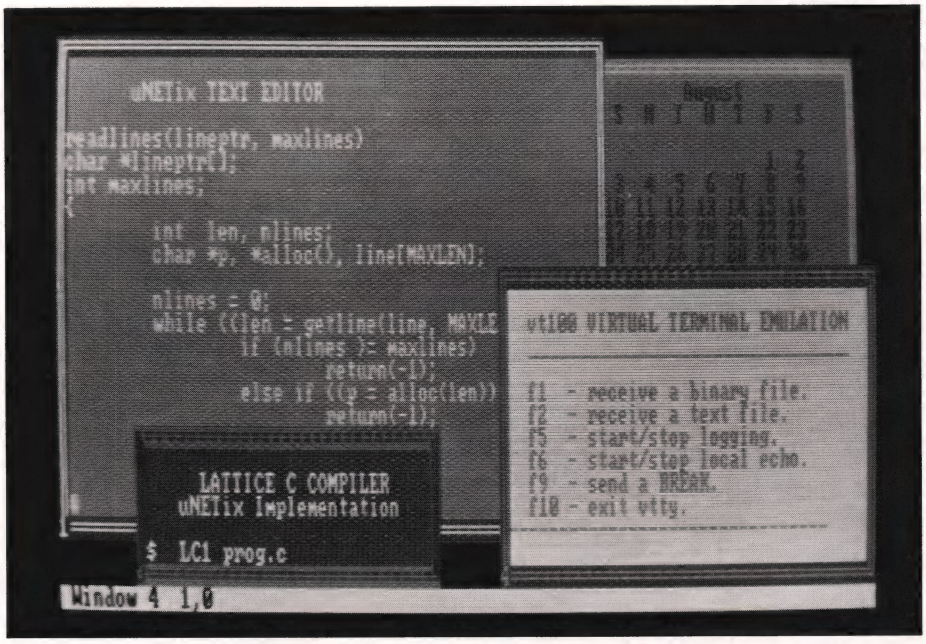
File Maintainance Listing (Listing Continued, text begins on page 24)

```
 9          RLEN ! BBLOCK ! ;          ( BBLOCK IS START OF FILE)
10
11 : RECORD ( RNO --- ADDR     RETURNS RECORD ADDRESS )
12          RNO !    1024 RLEN @ / REC/BLK !
13          RNO @ REC/BLK @ /MOD     ( POSITION IN BLOCK)
14          BBLOCK @ + BLOCK SWAP RLEN @ * + DUP RLOC ! ;
15 -->

Screen # 107
 0 ( NUMERIC CHECK ROUTINES - ?DIGIT ?NUMBER  )
 1 0 VARIABLE DFLAG          ( DIGIT FLAG)
 2
 3 : ?DIGIT          ( N --- F     IS CHARACTER A DIGIT? )
 4          DUP 47 > SWAP 58 < AND ;
 5
 6 : ?NUMBER         ( LOC --- F )
 7          0 DFLAG !          DUP 11 + SWAP DO
 8                   I C@ ?DIGIT IF 1 DFLAG !
 9                              ELSE
10                   I C@ 32 = I C@ 0= OR
11                              IF LEAVE ELSE I C@ 46 = 0=
12                              IF 0 DFLAG ! LEAVE
13                   THEN THEN THEN LOOP
14          DFLAG @ ;
15 -->

Screen # 108
 0 ( MISC ROUTINES              ASK  LEN  WAIT )
 1
 2 : ASK             ( --- F            PROMPT Y/N )
 3     '    ." (Y/N) " KEY 78 = 0= ;          ( TRUE IF YES )
 4
 5 : LEN    ( ADDR --- COUNT )   ( RETURN LENGTH OF STRING )
 6     255 0 DO DUP I + C@ 0= IF I LEAVE THEN
 7          LOOP SWAP DROP ;
 8
 9 : WAIT   ( --- )   ( WAIT FOR KEYPRESS )
10     CR    CR    ." ANY KEY TO CONTINUE"  KEY DROP ;
11
12 : INPUT      ( NUMBER INPUT )
13      PAD 11 EXPECT PAD LEN PAD + 32 SWAP C! PAD 1 - NUMBER ;
14 -->
15

Screen # 109
 0 ( FILE MAINT - PINIT )
 1 0 VARIABLE PHIGH          ( HIGH ENTRY ON THE FILE )
 2 0 VARIABLE FNO            ( FIELD NO                )
 3 0 VARIABLE FLOC           ( FIELD LOCATION IN RECORD      )
 4 0 VARIABLE PDONE          ( PAGE COMPLETE FLAG     )
 5 0 VARIABLE PLOC           ( PAGE TABLE LOCATION    )
 6 0 VARIABLE PMAX           ( NUMBER OF FIELDS ON THIS PAGE )
 7
 8 ( INITIALIZE THE PAGE <SCREEN> PROCESSING WITH THE       )
 9 ( LOCATION OF THE DESCRIPTION TABLE BUILT WITH PBUILD    )
10
11 : PINIT          ( TBLLOC --- )
12          DUP PLOC !        ( REMEMBER WHERE THE TABLE IS    )
13          C@ PMAX C!        ( SAVE MAX FIELD COUNT           )
14          0 PDONE !         0 FNO ! ;        ( CLEAR FLAGS   )
15 -->
```

(Continued on page 30)

SOFTWARE DEVELOPERS

LESS THAN $400 WILL TURN YOUR PC INTO A POWERFUL DEVELOPMENT WORKSTATION.



Until now, software developers who wanted the power of high-priced development systems had to pay a high price for it. There was no choice.

Now, there's a solution . . . whether you're developing programs to run on minicomputers or micros, and regardless of what target operating system you're using.

If an IBM® PC, PC XT™, or compatible is part of your development environment, the uNETix Software Development ToolKit will increase your productivity.

LANTECH SYSTEMS has put an end to the "no choice" problem with a collection of powerful software products which make your PC a versatile, but low cost development workstation.

The uNETix Software Development ToolKit contains:

- **uNETix SFS** — the powerful multi-tasking, **UNIX**™ compatible operating system, which is at the heart of the LANTECH SYSTEMS product line. The unique "**Window Management**" feature provided with uNETix allows easy integration of applications software and permits data to be moved between various window processes (such as the Emulator or Editor described below).
- **VT100**® Terminal Emulator (VTTY) — permits up-loading and down-loading of data between your PC and a larger mini or mainframe computer. This function, of course, means that larger-scale development tasks can be run on more powerful machines, while permitting you to process other tasks locally.
- **TEXT EDITOR** — allows powerful, full-screen editing at the workstation level or you can use your familiar mainframe editor through the terminal emulator. Several popular editors are available.

The LANTECH SYSTEMS uNETix Software Development ToolKit is the solution to your software development needs, at an affordable price of $399. Quantity discounts are available.

Optional Lattice® C Compilers are available from LANTECH SYSTEMS to develop programs locally to run under uNETix, or PC DOS™. Programs developed for uNETix can, of course, be easily ported to other UNIX environments. Lattice C Cross Compilers are also available for a variety of minicomputers.

To order your ToolKit, or to get more information contact:

LANTECH SYSTEMS
INCORPORATED

9635 WENDELL ROAD, DALLAS, TEXAS 75243
(214) 340-4932 EX. 200

uNETix is a trademark of Lantech Systems, Inc. IBM is a registered trademark of International Business Machines. UNIX is a trademark of Bell Laboratories, AT&T Technologies. VT100 is a trademark of Digital Equipment Corp. Lattice is a registered trademark of Lattice, Inc. PC DOS is a trademark of International Business Machines. XT is a trademark of International Business Machines. Minimum system configuration is IBM PC with 512K RAM, monochrome or color display. RS232 port required for VT100 connection.

Circle **no. 35** on reader service card.

File Maintainance Listing (Listing Continued, text begins on page 24)

```
Screen # 110
  0 ( FILE MAINT - PBUILD )
  1
  2 ( THIS BUILDS THE FILE & SCREEN DESCRIPTION THAT CONTROLS )
  3 ( THE WHOLE PROCESS  - ENTER WITH NUMBER OF FIELDS ON STACK )
  4
  5 : PBUILD <BUILDS DUP  C, 0 DO      ( NO_OF_FLDS --- )
  6          32 WORD HERE NUMBER DROP C,  ( ROW TO DISPLAY PROMPT )
  7          32 WORD HERE NUMBER DROP C,  ( COLUMN  "    "    " )
  8          32 WORD HERE NUMBER DROP C,  ( ROW FOR INPUT DATA )
  9          32 WORD HERE NUMBER DROP C,  ( COLUMN "      "     )
 10          32 WORD HERE NUMBER DROP C,  ( INPUT DATA LENGTH )
 11          32 WORD HERE NUMBER DROP C,  ( INPUT DATA TYPE )
 12          32 WORD HERE NUMBER DROP C,  ( DATA POSITION IN REC )
 13          34 WORD HERE C@ 1+ ALLOT     ( PROMPT TEXT TO DISPLAY )
 14   LOOP DOES> ;
 15 -->

Screen # 111
  0 ( FILE MAINT - FFIND  FDISPLAY  PDISPLAY          )
  1 : FFIND ( FNO --- FLOC          FIND FIELD DATA IN TABLE )
  2          PLOC @ 1+ FLOC ! -DUP IF
  3                   0 DO
  4                   FLOC @ 7 + DUP C@ + 1+ FLOC !
  5                   LOOP
  6          THEN FLOC @ ;
  7 : FDISPLAY ( FLOC --- )          ( DISPLAY FIELD'S PROMPT )
  8          DUP DUP C@ SWAP 1+ C@ XY 7 + COUNT TYPE
  9          FLOC @ 2 + C@ FLOC @ 3 + C@ XY FLOC @ 4 + C@
 10          0 DO 95 EMIT LOOP ;
 11 : PDISPLAY ( --- )  ( DISPLAY FULL PAGE OF PROMPTS )
 12          CLS PMAX C@ 0 DO
 13                   I FFIND FDISPLAY
 14          LOOP ;
 15 -->

Screen # 112
  0 ( FILE MAINT - ?FDATA   FDATA )
  1 ( ?FDATA CHECKS TO SEE IF FIELD GETS DATA - COULD BE )
  2 ( DISPLAY ONLY )
  3
  4 : ?FDATA ( FLOC --- FLOC TRUE  or FLOC --- FALSE )
  5          DUP 4 + C@ 0= IF DROP 0 ELSE 1 THEN ;
  6
  7 : FDATA  ( FNO --- LOC LENGTH TYPE or  --- 0 if non-data )
  8          FFIND ?FDATA IF
  9                   DUP 2+ C@ SWAP 3 + C@ XY ( POSITION CURSOR)
 10                   RLOC @ FLOC @ 6 + C@ +  ( FIELD LOCATION )
 11                   FLOC @ 4 + C@           ( FIELD LENGTH )
 12                   FLOC @ 5 + C@           ( FIELD TYPE )
 13          ELSE     0        ( NOT A DATA FIELD )
 14   THEN ;
 15 -->


Screen # 113
  0 ( FILE MAINT - FGET      FTEXT    FNUM )
  1 0 VARIABLE FTYPE
  2 : FTEXT ( LOC OLEN PLEN --- ) ( GET TEXT TYPE DATA )
  3          ROT ROT 2DUP BL FILL DROP
  4          PAD SWAP ROT CMOVE ;
  5 : FNUM 2DROP PAD DUP 11 + SWAP DO I C@ 0= IF 32 I C! THEN LOOP
```

```
 6       PAD ?NUMBER IF  PAD 1- NUMBER ELSE 0 0 THEN ;
 7 : FGET  FDATA -DUP IF   ( FNO --- )
 8    FTYPE ! PAD OVER EXPECT PAD LEN  -DUP IF
 9                  FTYPE @ 9 = IF FTEXT THEN
10                  FTYPE @ 1 = IF FNUM DROP SWAP C! THEN
11                  FTYPE @ 2 = IF FNUM DROP SWAP ! THEN
12                  FTYPE @ 3 = IF FNUM DROP SWAP ! THEN
13                  FTYPE @ 4 = IF FNUM ROT 2! THEN
14                  FTYPE @ 5 = IF FNUM ROT 2! THEN
15          ELSE 2DROP THEN THEN ;                      -->

Screen # 114
 0 ( FILE MAINT - PUT DATA TO SCREEN  -  FPUT   .$   )
 1 : .$ SWAP OVER DABS     ( DISPLAY MONEY FIELD )
 2         <# # # 46 HOLD #S SIGN #> TYPE SPACE ;
 3
 4 : FPUT ( FNO --- )       ( DISPLAY DATA FIELD )
 5         FDATA -DUP IF FTYPE !
 6                  FTYPE @ 9 = IF TYPE THEN
 7                  FTYPE @ 1 = IF SWAP C@ SWAP .R THEN ( CHAR)
 8                  FTYPE @ 2 = IF SWAP @ SWAP .R THEN ( SINGLE )
 9                  FTYPE @ 3 = IF DROP @ 0 .$ THEN ( DOLLARS )
10                  FTYPE @ 4 = IF SWAP 2@ SWAP D.R THEN
11                  FTYPE @ 5 = IF DROP @ .$ THEN  ( D$'S )
12         THEN ;
13
14 -->
15

Screen # 115
 0 ( FILE MAINT -  PGET    PPUT )
 1 ( SET RLOC <RECORD LOCATION> PRIOR TO CALLING EITHER OF )
 2 ( THESE WORDS )
 3
 4 : PGET ( --- )  ( GET ALL THE ITEMS ON A PAGE )
 5         PLOC @ C@       ( GET THE FIELD COUNT )
 6            0 DO I FGET LOOP ;
 7
 8 : PPUT ( --- )   ( DISPLAY ALL ITEMS FROM A RECORD )
 9         PLOC @ C@       ( # FIELDS )
10           0 DO I FPUT LOOP ;
11
12 -->
13
14
15

Screen # 116
 0 ( FILE MAINT - PNEXT    PADD    PLIST  )
 1
 2 : PNEXT          ( NEXT RECORD POSITION TO ADD )
 3         PHIGH @ 1 + DUP PHIGH ! RECORD ;
 4
 5 : PADD           ( ADD OR CHANGE A RECORD )
 6         PDISPLAY PGET 0 ;
 7
 8 : PLIST          ( LIST SELECTED PORTION OF RECORDS )
 9         CLS
10         PHIGH @ 1 + 1 DO
11                  I 3 .R SPACE
12                  I RECORD 20 TYPE CR  ( NAMES FOR NA TEST )
13                  I 23 MOD 0= IF WAIT CLS THEN
14         LOOP WAIT 0 ;
15 -->
```

(Continued on next page)

File Maintainance Listing (Listing Continued, text begins on page 24)

```
Screen # 117
  0 ( FILE MAINT  -  PSELECT  PEND )
  1
  2 : PSELECT         ( PICK A RECORD FOR FURTHER ACTION )
  3          CLS CR ." ENTER RECORD NO.  "
  4          INPUT DROP
  5          DUP PHIGH @ > IF ." NO SUCH RECORD" DROP 0
  6                    ELSE    RECORD DROP 1 THEN ;
  7
  8 : PEND            ( TERMINATE THIS TASK )
  9          CR ." SURE " ASK IF
 10                    PHIGH @ 0 RECORD ! UPDATE
 11                    CLS ." SAVING FILES" CR
 12                    FLUSH ." BYE" CR 1
 13          ELSE 0 THEN ;
 14
 15 -->
```

```
Screen # 118
  0 ( FILE MAINT - PMENU )
  1 : PMENU CLS 6 10 XY ." FILE MAINTAINANCE"
  2          10 1 XY ." 1 - ADD" CR
  3          12 1 XY ." 2 - LIST" CR
  4          14 1 XY ." 3 - DISPLAY A RECORD" CR
  5          16 1 XY ." 4 - CHANGE A RECORD" CR
  6          18 1 XY ." 9 - QUIT" CR
  7          20 10 XY ." SELECT ONE OF ABOVE " KEY
  8    DUP 49 = IF DROP 0 PNEXT PADD
  9    ELSE DUP 50 = IF DROP PLIST
 10    ELSE DUP 51 = IF DROP PSELECT IF PDISPLAY PPUT WAIT 0 THEN
 11    ELSE DUP 52 = IF DROP PSELECT IF PDISPLAY PPUT PGET 0 THEN
 12    ELSE DUP 57 = IF DROP PEND
 13    ELSE DROP 0
 14    THEN THEN THEN THEN THEN ;
 15
```

```
Screen # 119
  0 ( FILE MAINT TEST - NAME AND ADDRESS APPLICATION )
  1
  2 5 PBUILD NAPAGE
  3 6   10  0   0   0   0   00 MAIL LIST"
  4 10  1   10  15  20  9   00 NAME......."
  5 12  1   12  15  20  9   20 ADDRESS...."
  6 14  1   14  15  20  9   40 CITY STATE."
  7 16  1   16  15  5   9   60 ZIP CODE..."
  8
  9 : NA    ( TASK CALL - REMEMBER TO SET 0 RECORDS AT BLOCK )
 10     130 65 IREC     ( SET START BLOCK & RECORD LENGTH )
 11     0 RECORD @ PHIGH !  NAPAGE PINIT
 12     BEGIN PMENU UNTIL ;
 13
 14
 15
```

```
Screen # 120
  0 ( FILE MAINT SAMPLE 2 )
  1
  2 8 PBUILD M.PAGE
  3 3   5   0   0   0   0   0 MOLD FILE MAINT"
  4 5   1   5   15  5   9   0 VENDOR"
  5 7   1   7   15  10  9   5 VENDOR NO"
  6 9   1   9   15  20  9   15 CATAGORY"
  7 11  1   11  15  24  9   35 DESCRIPTION"
  8 13  1   13  15  6   3   59 G PRICE"
  9 15  1   15  15  6   3   61 COST"
 10 17  1   17  15  6   3   63 RETAIL"
 11
 12 : MOLD
 13      131 80 IREC 0 RECORD @ PHIGH !
 14     M.PAGE PINIT BEGIN PMENU UNTIL ;
 15
```

End Listing

Forth and the Fast Fourier Transform

by Joe Barnhart

Long the province of the scientist and the engineer, the Fast Fourier Transform (or FFT) has found its way into a fantastic number of applications. Heat transfer, medical imaging, and sales data analysis are just a few of the FFT's many capabilities. In this article, I use the FFT to analyze stock market data and detect cyclic trends.

This FFT package is written in the Forth language. Novices to Forth often wonder how any serious work can be done without floating-point numbers. Forth programmers can benefit from seeing how the FFT was converted from an algorithm that is usually performed with floating-point numbers into one that is suitable for Forth's fixed-point math. Not surprisingly, the fixed-point version will run rings around most floating-point versions in terms of execution speed.

This program was written with transportability in mind. Because this program is written in the new Forth-83 Standard version, it should be transportable to many different systems. I used a Forth system from Laboratory Microsystems, Inc. (LMI), that adheres to the new standard and runs under the CP/M-68K operating system. My hardware is a Sage II with an 8-MHz Motorola 68000 processor, but the same code runs without modification on LMI's IBM PC/Forth and Z80 Forth (also Forth-83 Standard).

Those accustomed to using floating-point arithmetic often bemoan Forth's fixed-point orientation. It's remarkable, though, how much you can do—and how efficiently you can do it—using only fixed-point math.

Joe Barnhart, 522 Talbot Avenue, Albany, CA 94706.

A Little Theory

What exactly does the Fourier Transform do? Put simply, it relates time to frequency. For example, imagine a man beating two rocks together at a steady rate. If you graph the sound made by the rocks versus time, you get the curve shown in Figure 1 (page 36). Every fourth point shows the sound made as the rocks came together. On the other hand, you might represent the event by the graph in Figure 2 (page 36). In this graph, you can see that the axes are labeled in terms of sound versus frequency, and the only response is at 0.25 Hz (or 0.25 beats/second). These two graphs are equivalent ways of looking at the same process.

As a more realistic example, consider the tone source for an electronic music synthesizer. It can produce many different musical timbres by altering the shape of the tone waveform. Figure 3 (page 36) shows a square wave. The sound that the wave represents is harsh, sharp, and buzzing because it is composed of many high frequencies. If you look at the square wave with a tool called a spectrum analyzer, you see the pattern in Figure 4 (page 36). The square wave is composed of many different pure sine waves. The Fourier Transform of the square wave yields exactly the same pattern as does the spectrum analyzer.

The Discrete Fourier Transform (DFT) is a simple extension of the Fourier Transform that handles data that is represented as points instead of continuous curves (Figure 5, page 36). The Fast Fourier Transform (or FFT)

is a quick method of computing a Discrete Fourier Transform.

To gain speed in computing a Fourier Transform, the FFT algorithm requires that the number of input samples (data points) be an integral power of 2 (e.g., 64, 128, 256). Also, the number of time samples (input to the FFT) and the number of frequency samples (output from the FFT) are the same. These restrictions are a small price to pay for the tremendous increase in speed of the FFT over the DFT algorithm.

How much faster is the FFT? In theory, the time required to calculate a DFT is proportional to N^2, where N is the number of pointers in the input data. The FFT requires time proportional to only $N*\log_2 N$. For example, say a DFT takes one second to calculate a 128-point transform. The FFT of the same data on the same computer can be calculated in only 55 milliseconds!

For the technically inclined, the FFT algorithm presented here is a radix-2, decimation-in-time version. This program takes a complex input array and returns a complex output array—I haven't taken advantage of special symmetry conditions to make the algorithm run faster. On the other hand, this is the most general use of FFT algorithms because it makes no assumptions about the input data.

The Whole Banana

For clarity and modularity, this program is broken in to four distinct modules: a complex math package, a one-dimensional array package, the FFT algorithm itself, and the demonstration application.

Complex Numbers

To implement an FFT program, the computer must be able to use complex - 1. The value of a complex number is derived from the square root of negative one. I won't dwell on the theory of complex numbers here: instead, I'll discuss this implementation of a complex-number package in Forth.

Complex numbers occupy twice as much space as normal integers in Forth. Each complex number is composed of two distinct parts, called the "real" and the "imaginary" parts of the number. In my implementation, the real part is always on the top of the stack and the imaginary part underneath (Figure 6, page 36). Routines are defined to add, subtract, divide, and multiply complex numbers (X+, X–, X*, X/). Other functions unique to complex numbers are magnitude (|X|) and conjugation (X').

The complex arithmetic functions X* and X/ operate on scaled numbers. For instance, I often scale numbers so that 10000 represents the number 1.0000. Whenever two scaled numbers are multiplied, the result must be divided by the scale factor. Similarly, after dividing two scaled numbers, the result must be multiplied by the scale factor. To increase accuracy, intermediate products are maintained in double precision, as in the Forth word */. Addition and subtraction of scaled numbers is identical to the same operations performed on numbers without scaling.

One-Dimensional Array

A one-dimensional array is often referred to as a vector in math and engineering circles. This vector definition is a general-purpose one and can serve many programs other than this FFT. The vector data structure (Figure 7, page 37) contains the number of vector elements, the size of each element (in bytes), the scale factor of the vector, and the number of bytes of storage used by the data area of the vector.

Independence from machine word size is always beneficial when moving an application from one computer to another. This array is indexed by element number rather than by byte offset. A complex number requires four bytes (in a 16-bit Forth system) and an integer 2, but the seventh element of a complex array is accessed in exactly the same way as the seventh element of an integer array. Because the size of each entry is stored with the array, it is a simple matter for the indexing word, [I], to determine the byte offset from the element number.

FFT Routine

This algorithm was taken from a Fortran program (reference 2, page 40). It is a simple, direct method of calculating the Fourier Transform of discrete points. The program consists of two parts: a bit reversal and the FFT "kernel" itself.

This program can compute both the

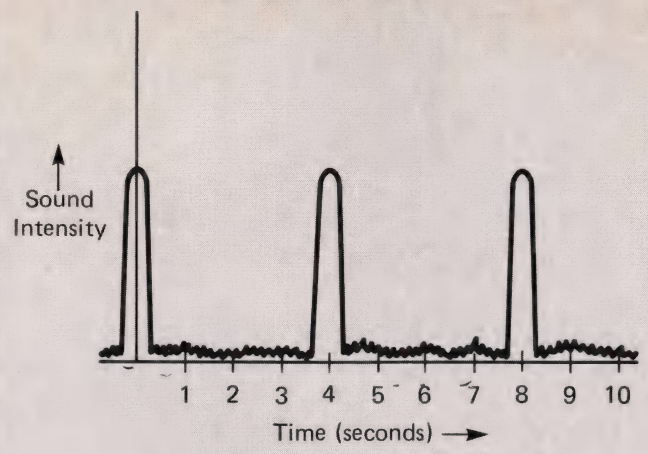


Figure 1 A sound is made every 4 seconds.

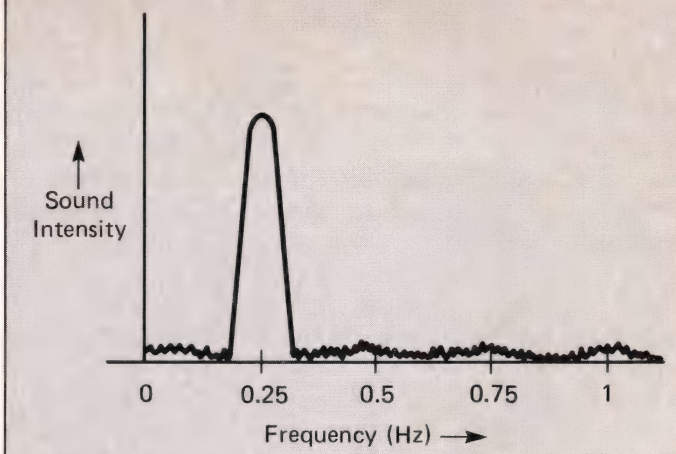


Figure 2 Another way of graphing the same sound.

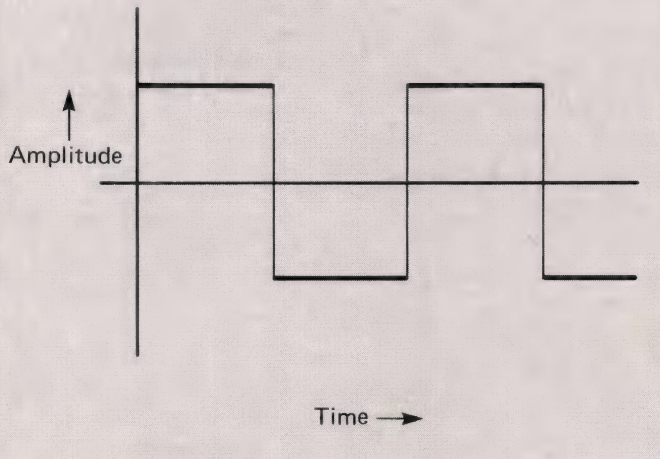


Figure 3 A "square" wave.

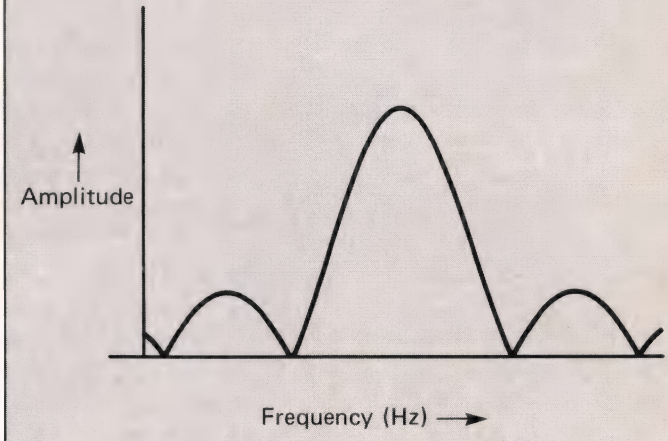


Figure 4 The frequency content of the square wave.

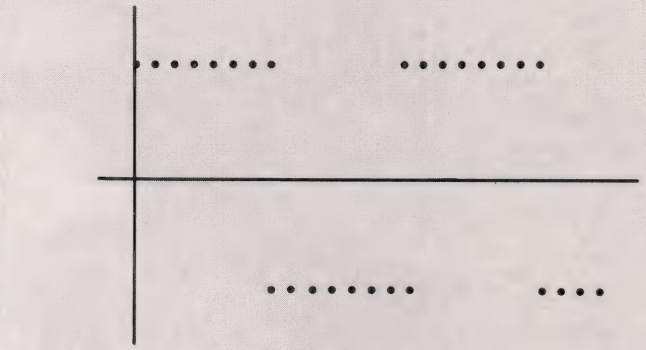


Figure 5 "Discrete" data in the form of a square wave.

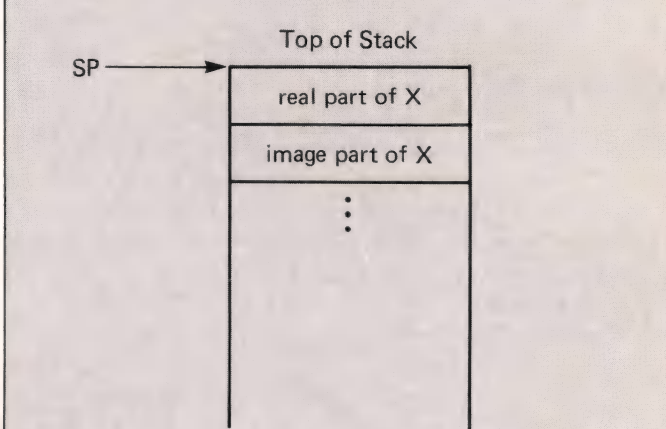


Figure 6 Complex number representation on the stack.

FFT and the Inverse FFT (IFFT). The intended direction of the transform is held in the variable DIR (true = inverse transform). The address of the data array and the number of elements in the array are held by variables, because they are referred to at all levels of nesting.

Bit reversal reorders the vector in preparation for the computation of the transform. This "shuffling" of the data is one reason why the FFT is so much quicker than the DFT. Figure 8 (page 37) shows a small data array and how it is reordered by the routine BIT-REVERSE.

After ordering of the data, the transform itself is computed as a loop within a loop. The inner loop (INNER-LOOP) is factored out of the outer loop (FFT-KERNEL) for readability. At each point in the inner loop, a computation nicknamed "butterfly" is performed. Looking at the diagram of the butterfly computation (Figure 9, page 37), you can see how the name was chosen. Notice that direction of the FFT is tested in the routine BFLY. One of the few differences between a FFT and its inverse is that the FFT is scaled (divided) by the number of points in the data array.

To free myself from the tyranny of trignometric functions, I created a table, WTAB, that contains the sines and cosines needed for FFT computations. WTAB is in the same form as a complex vector, so you can access it with the same words used for vector operations (especially [I]). For FFT's, only the sine and cosine for angles of pi/N are needed, where N is the number of data points. This table can handle FFT's of up to 128 points and is easily extended for larger FFT's.

The Application

The hardest part of having an FFT program is deciding what to use it on. With so many potential applications, it's hard to settle on one for demonstration purposes. I originally wrote this package for use with image processing applications, but it is so useful I'm finding new applications daily—such as looking at stock market trends.

Few of us doubt that stocks move in cyclical patterns. In screen 35 (page 49) I've tabulated the price of IBM common stock by week for all of 1983. The price is normalized to 100—for instance, a price of $117 and 7/8 is entered as 11788 ($117.88). Just looking at the raw data, we can see some evidence of cycles (Figure 10, page 38). Some cycles in this data are masked by noise—hidden from the casual observer.

A few notes about the data used in this example are in order. Notice that I have only 52 data points to transform. The FFT only works with a number that is an integral power of 2 (such as 64). To use the FFT in this case, I've filled the array with zeros up to 64 points. When using the FFT on this data, some

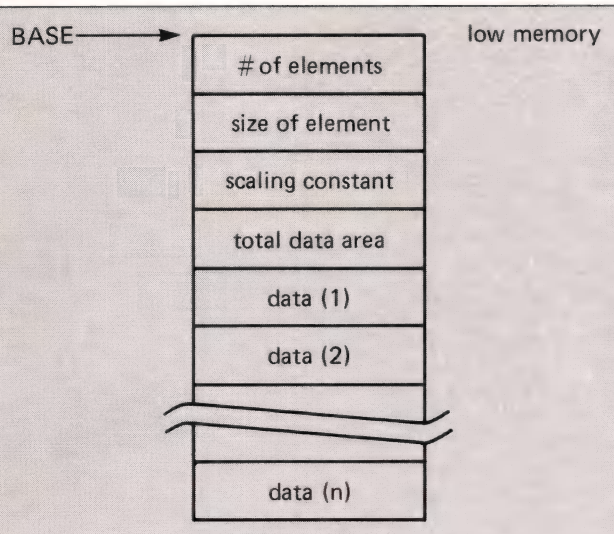


Figure 7
Vector data structure.

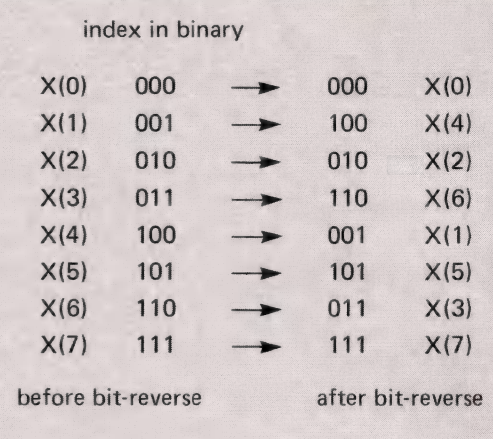


Figure 8
Bit-reversal of a small array.

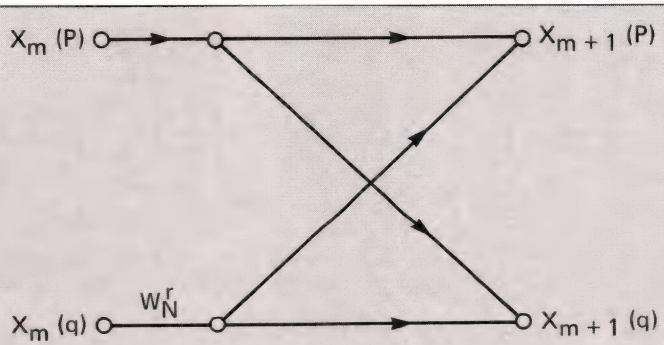
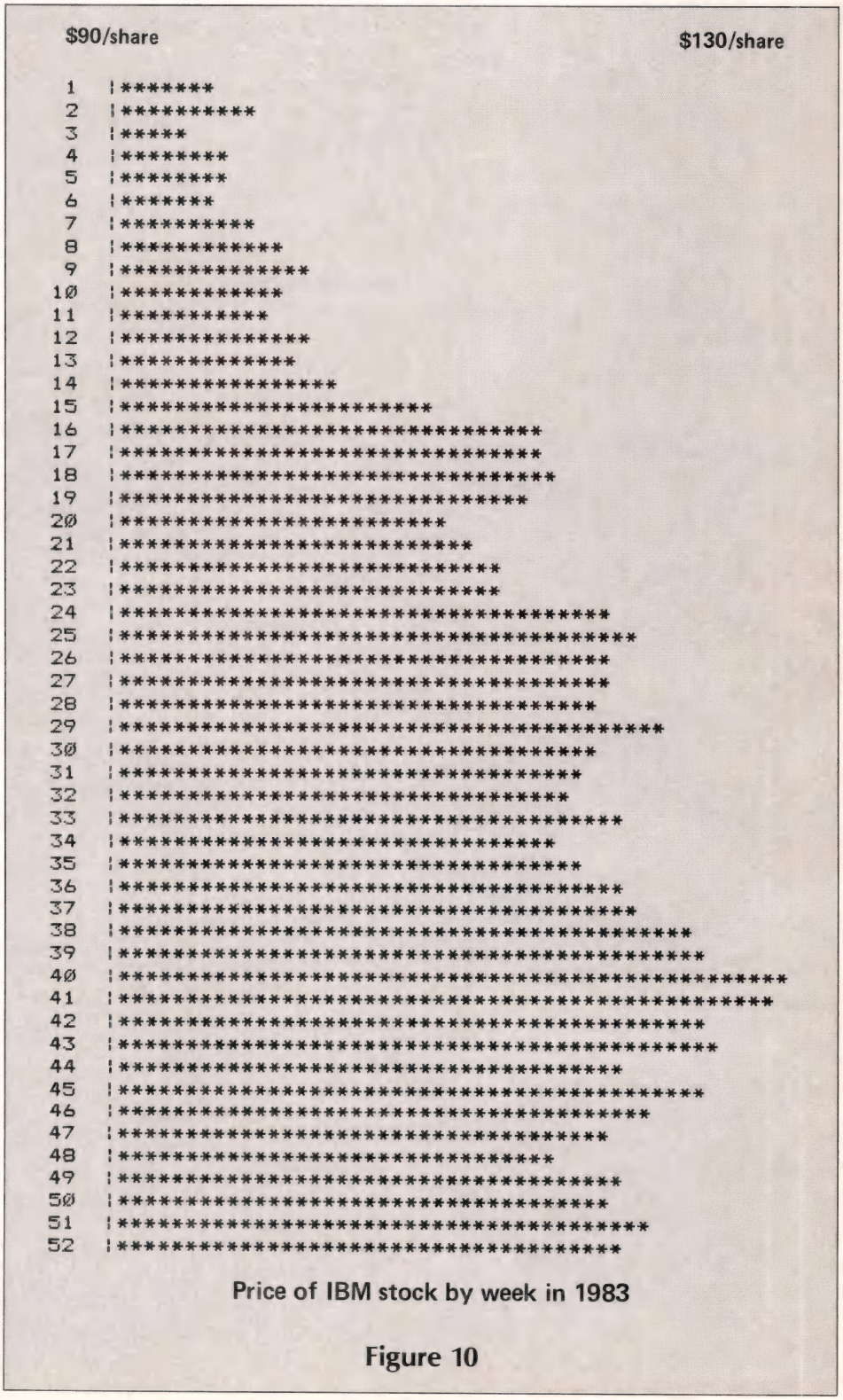


Figure 9 Flow graph of a "butterfly" computation.
(Source: *Digital Signal Processing,* Oppenheim & Schaeffer, Prentice-Hall, pg. 296, fig. 6.9)

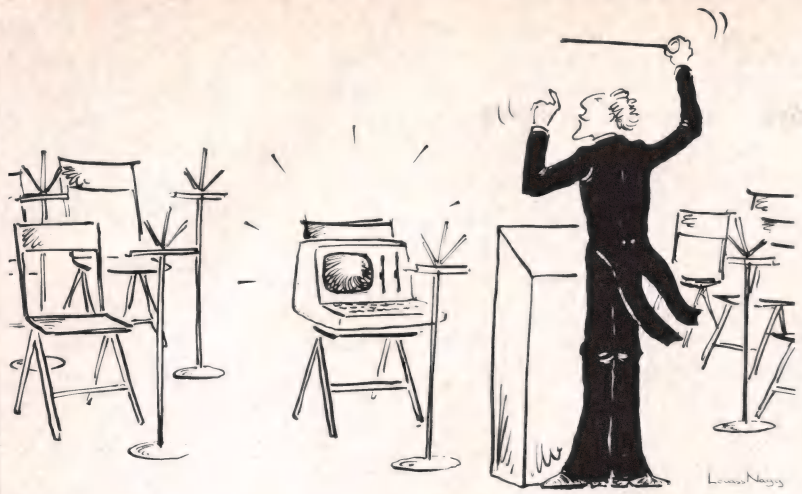
phantom responses can creep in because of the zeros. I've reduced the errors by employing a special technique called windowing the data—a process that reduces the sharpness of the data and hence the phantom responses (reference 1, page 40).

Performing a Fourier Transform on this stock data, we get a plot of all cycles in the data and their relative sizes (Figure 11, page 40). Even after disregarding normal endpoint effects (indices 1 . . 9 and 10 . . 14 on Figure 11), a few cycles are left. One curious cycle is at index 24, which corresponds to an event that repeats every 20 months or so. Along with their knowledge of the company, some stock chartists use the FFT to help forecast the future price of a stock, given its history.



Price of IBM stock by week in 1983

Figure 10



Would you hire an entire band when all you need is one instrument? Of course not.

So why use a whole orchestra of computers when all you need is one to develop software for virtually any type of micro-processor?

The secret? Avocet's family of cross-assemblers. With Avocet cross-assemblers you can develop software for practically every kind of processor — *without having to switch to another development system along the way!*

Cross-Assemblers to Beat the Band!

Development Tools That Work

Avocet cross-assemblers are fast, reliable and user-proven in over 4 years of actual use. Ask NASA, IBM, Xerox or the hundreds of other organizations that use them. Every time you see a new microprocessor-based product, there's a good chance it was developed with Avocet cross-assemblers.

Avocet cross-assemblers are easy to use. They run on almost any personal computer and process assembly language for the most popular microprocessor families.

Your Computer Can Be A Complete Development System

Avocet has the tools you need to enter and assemble your soft-ware and finally cast it in EPROM:

VEDIT Text Editor makes source code entry a snap. Full-screen editing plus a TECO-like command mode for advanced tasks. Easy installation - INSTALL program supports over 40 terminals and personal computers. Customizable keyboard layout. CP/M-80, CP/M-86, MSDOS, PCDOS $150

EPROM Programmers let you program, verify, compare, read, display EPROMS but cost less because they communicate through your personal computer or terminal. No personality modules! On-board intelligence provides menu-based setup for 34 different EPROMS, EEPROMS and MPUs (40-pin devices require socket adaptors). Self-contained unit with internal power supply, RS-232 interface, Textool ZIF socket. Driver software (sold separately) gives you access to all programmer features through your computer, lets you download cross-assembler output files, copy EPROM to disk.

Model 7228 Advanced Programmer — Supports all PROM types listed. Superfast "adaptive" programming algorithm programs 2764 in 1.1 minutes.

Model 7128 Standard Programmer — Lower-cost version of 7228. Supports all PROM types except "A" versions of 2764 and 27128. Standard programming algorithm programs 2764 in 6.8 minutes.

Avocet Cross-assembler	Target Microprocessor	CP/M-80	CP/M-86 IBM PC, MSDOS**
XASM04 *NEW*	6804	$ 250.00	$ 250.00
XASM05	6805	200.00	250.00
XASM09	6809	200.00	250.00
XASM18	1802/1805	200.00	250.00
XASM48	8048/8041	200.00	250.00
XASM51	8051	200.00	250.00
XASM65	6502/65C02	200.00	250.00
XASM68	6800/01, 6301	200.00	250.00
XASM75	NEC 7500	500.00	500.00
XASM85	8085	250.00	250.00
XASM400	COP400	300.00	300.00
XASMF8	F8/3870	300.00	300.00
XASMZ8	Z8	200.00	250.00
XASMZ80	Z80	250.00	250.00
XMAC682 *NEW*	68200	595.00	595.00
XMAC68K *NEW*	68000/68010	595.00	595.00

Model 7956 and 7956-SA Gang Programmers — Similar features to 7228, but program as many as 8 EPROMS at once. 7956-SA stand-alone version copies from a master EPROM. 7956 lab version has all features of stand-alone plus RS-232 interface.

EPROM: 2758, 2716, 2732, 2732A, 2764, 2764A, 27128, 27128A, 27256, 2508, 2516, 2532, 2564, 68764, 68766, 5133, 5143. **CMOS:** 27C16, 27C32, 27C64, MC6716. **EEPROM:** 5213, X2816A, 48016, I2816A, 5213H. **MPU (w/adaptor):** 8748, 8748H, 8749, 8749H, 8741, 8742, 8751, 8755.

7228	**Advanced Programmer**	**$ 549**
7128	**Standard Programmer**	**429**
7956	**Laboratory Gang Programmer**	**1099**
7956-SA	**Stand-Alone Gang Programmer**	**879**
PDV	**Driver Software**	**95**
481	**8748 Family Socket Adaptor**	**98**
511	**8751 Socket Adaptor**	**174**
755	**8755 Socket Adaptor**	**135**
CABLE	**RS-232 Cable (specify gender)**	**30**

HEXTRAN Universal HEX File Converter — Convert assembler output to other formats for downloading to development systems and target boards. Also useful for examining object file, changing load addresses, extracting parts of files. Converts to and from Intel, Motorola, MOS, RCA, Fairchild, Tektronix, TI, Binary and HEX/ASCII Dump formats. For CP/M, CP/M-86, MSDOS, PCDOS $250

Ask about UNIX.

AVOCET'S SUPERB 68000 CROSS-ASSEMBLER — With exhaustive field testing completed, our 68000 assembler is available for immediate shipment. XMAC68K supports Motorola standard assembly language for the 68000 and 68010. Macros, cross-reference, structured assembly statements, instruction optimization and more. Linker and librarian included. Comprehensive, well-written manual. XMAC682 for MK68200 has similar features.

Call us toll-free for some straight talk about development systems.

1-800-448-8500

(in the U.S. Except Alaska and Hawaii)

VISA and Mastercard accepted. All popular disc formats now available - please specify. Prices do not include shipping and handling - call for exact quotes. OEM INQUIRIES INVITED.

*Trademark of Digital Research **Trademark of Microsoft

AVOCET SYSTEMS INC.™

DEPT. 984-DDJ
804 SOUTH STATE STREET
DOVER, DELAWARE 19901
302-734-0151 TELEX 467210

Circle **no. 5** on reader service card.

Other Uses

My original use for this FFT package involved the field of image processing. I used this one-dimensional FFT to build a two-dimensional FFT and performed filtering and restoration on digitized images.

Other uses for this transform abound. If you have a microphone and can digitize its output, you can use the FFT to estimate the frequency response of your stereo system. Feed your stereo with a white noise source and sample the output from the microphone at intervals of 25 microseconds or so. The FFT of the microphone's output indicates the frequency response of your stereo.

An exciting use of the FFT is in the field of shape recognition. Using a touch tablet or digitizer, draw a particular shape (such as a square) over and over again, in various sizes and orientations. Take the (x, y) pairs from the touch tablet and run them through the FFT. From the output of the FFT, coefficients can be calculated (called "Fourier descriptors") that are unique for each shape processed. Recognition of shapes and characters is much easier with these descriptors because they are invariant to the size, rotation, and position of the shape on the tablet.

References

1. *Theory and Application of Digital Signal Processing*, Lawrence Rabiner and Bernard Gold, Prentice Hall, Englewood Cliffs, NJ. A good second book on the FFT and its uses. You should be familiar with the rudiments of the FFT before tackling this book.
2. *Digital Image Processing*, Rafael Gonzales and Paul Wintz, Addison-Wesley, Reading, MA. Excellent book on all aspects of image processing, including the two-dimensional Fourier Transform and the material on Fourier descriptors.
3. *The Fourier Transform and its Applications*, Ronald Bracewell, McGraw-Hill, New York, NY. A good first text on the Fourier Transform, this book requires first-year calculus for full understanding.

Glossary

These entries are stack diagrams and definitions of the Forth words defined in the FFT program. I've pointed out non-Forth-83 words where I've used them. Stack contents are shown before execution (left side) and after execution (right side). The top of the stack is on the right side of each list. The words are listed in order of their definition.

D2* (dbl dbl --- dbl)
Multiply a double-precision number by 2. Used by the square-root routine.

EASY-BITS,
2'S-BIT,
1'S-BIT
Captive definitions used only by the double-precision square-root routine.

SQRT (dbl --- sgl)
Take the square root of a double-precision number and leave the single-precision result on the stack. This word uses the double-precison extension word DU< (which is Forth-83 but may not be on your system). It can be defined as:

```
: DU< ROT SWAP 2DUP U<
      IF 2DROP 2DROP -1
      ELSE = >R U< R>
      AND
      THEN ;
```

SQR (sgl --- dbl)
Square the single-precision number and leave the double-precision result on the stack. This definition

```
 1  |**************************************************************************
 2  |**************************************************************************
 3  |**************************************************************************
 4  |**************************************************************************
 5  |********************************
 6  |************************************
 7  |**************************************************************************
 8  |******************************************************
 9  |************************************
10  |**
11  |**********************
12  |**********************
13  |**********************
14  |*********
15  |**
16  |*******
17  |**
18  |*******
19  |********
20  |*******
21  |*
22  |***
23  |*****
24  |**********   20 cycles/year fluctuation
25  |****
26  |*
27  |*
```

FFT of IBM stock price
Figure 11

uses the word M*, which *is not* Forth-83 Standard. M* takes two single-precision arguments and leaves a signed, double-precision product on the stack. An equivalent Forth-83 definition is:

```
: M* OVER OVER XOR >R
        ABS SWAP ABS UM*
     R> 0< IF DNEGATE
     THEN ;
```

X@ (addr --- cmplx)
Fetch a complex number from address and leave it on the stack.

X! (cmplx addr ---)
Store a complex number at address.

XVARIABLE (---)
Create a named entity that, when executed, leaves the address of a complex number storage location on the stack.

XCONSTANT (cmplx ---)
Create a named entity that, when executed, leaves a complex number on the stack.

XDUP (cmplx --- cmplx cmplx)
Duplicate the complex number on the stack.

XSWAP (cmplx2 cmplx1 --- cmplx1 cmplx2)
Swap the two complex numbers on the stack.

XDROP (cmplx ---)
Drop one complex number from the stack.

XOVER (cmplx2 cmplx1 --- cmplx2 cmplx1 cmplx2)
Copy the complex number under the top of the stack onto the top of the stack.

X2DUP (cmplx2 cmplx1 --- cmplx2 cmplx1 cmplx2 cmplx1)
Duplicate two complex numbers on the stack, keeping their relative positions.

X+ (cmplx2 cmplx1 --- cmplx2+1)
Add two complex numbers and leave the complex result on the stack.

X– (cmplx2 cmplx1 --- cmplx2–1)
Subtract the top complex number from the one beneath and leave the complex result.

X2/ (cmplx --- cmplx/2)
Divide the complex number on the stack by 2 and leave the complex result.

| X |^2 (cmplx --- dbl)
Leave the magnitude squared of the complex number on the stack. The magnitude squared is not a complex number but a double-precision integer.

| X | (complx --- sng)
Leave the single-precision magnitude of the complex number on the stack.

X' (cmplx --- cmplx)
Take the complex conjugate of the complex number on the stack and leave the complex result. (Change the sign of the imaginary part.)

X0= (cmplx --- flag)
Test the complex number on the stack for equivalence with zero and leave the flag on the stack.

X* (cmplx2 cmplx1 scale --- cmplx2*1/scale)
Multiply two complex numbers and then divide the complex result by the single-precision scaling constant, leaving a complex result on the stack.

X*! (addr2 addr1 n ---)
Both addresses point to complex numbers. Multiply the two complex numbers and divide by the single-precision scaling constant. Store the complex result into the area pointed to by addr1.

X/ (cmplx2 cmplx1 scale --- cmplx2/1 * scale)
Divide the top complex number into the one beneath and then multiply the complex result by the single-precision scaling constant, leaving a complex result on the stack.

VECTOR (order scale size ---)
Create a named entity that leaves its address when executed. The size of the data area is determined by the number of entries (order) and the size of each (size). The single-precision scaling constant associated with the data is saved in the structure.

&ORDER (vector --- order)
Given the address of a vector, leave the single-precision number of entries on the stack.

&ESIZE (vector --- size)
Given the address of a vector, leave the single-precision size of each element on the stack.

&SCALE (vector --- scale)
Given the address of a vector, leave the single-precision scaling constant of the data on the stack.

&SIZE (vector --- tot_size)
Given the address of a vector, leave the single-precision total number of bytes used by the data area of the vector.

[I] (indx vector --- addr)
Return the address of the element number that corresponds to index. If the requested element is out of bounds, generate an error and halt the program. All vectors are presumed to start with element one.

XVECTOR (order scale ---)
When executed, create a complex vector (see VECTOR) with order elements and set the scaling constant to scale.

X@V (indx vector --- cmplx)
Retrieve the element from the complex vector and leave the complex result on the stack.

X!V (cmplx indx vector ---)
Store the complex element into the complex vector.

10K (--- 10000)
Scaling constant for the trignometric constants in WTAB.

DIR (--- addr)
Variable that contains a flag that represents the direction of the transform—forward (false or 0) or inverse (true or –1).

VEC (--- addr)
Variable that contains the address of the vector undergoing transformation.

N (--- addr)
Variable that contains the size (or order) of the transformation. The same as the number of points in the data array.

U (--- addr)
Complex variable used only be the FFT routine.

W (--- addr)
Another complex variable used only by the FFT routine.

V[I][J] (j i --- addrj addri)
Given two indices, compute the address of the elements in the current vector (held by VEC). Leave the addresses on the stack.

XSWAPV (j i ---)
Swap the two complex numbers in the vector (held by VEC) pointed to by i and j.

BIT-REVERSE (---)
Reorder the complex vector (held by VEC) in preparation for the

FFT. For an example of a bit-reversed vector, see the text.

BFLY (j i ---)
Perform a butterfly computation on elements i and j of the vector in VEC. The butterfly is the form:
X[i] = X[j] – X[i]*U
X[j] = X[j] + X[i]*U
If the direction is inverse, then the results are divided by 2.

2** (sgl --- sgl)
Raise 2 to the power of the single-precision number on the stack. Leave the single-precision result. WSIZE lets this definition work on 16- or 32-bit systems.

LOG2 (sgl --- sgl)
Leave the logarithm (base 2) of the single-precision number on the stack. Again WSIZE is for portability.

WTAB (---)
A complex vector that holds the sines and cosines for the angles of the form pi/*N*, where *N* is the number of data points (order of the FFT).

INNER-LOOP (start incr ---)
Perform butterflies on the vector (in VEC) beginning at index "start" and adding "incr" each time. The butterfly is performed between index "I" and "I + incr/2," where I is the loop counter.

FFT-KERNEL (---)
Outer loop of the FFT program. Each loop sets the value of W and U and controls the action of the inner loop.

INIT (vector ---)
Initializes the variable VEC to contain the address of the vector and *N* to contain the number of elements.

DO-VECTOR (vector ---)
Perform a complete FFT or IFFT on the vector on the top of the stack.

FFT (vector ---)
Initialize the direction of the transform (variable DIR) and call DO-VECTOR.

IFFT (vector ---)
Initialize the direction of the transform and call DO-VECTOR.

DDJ

Fourier Transform Listing (Text begins on page 34)

Screen # 6
```
( Load screen )

FORTH DEFINITIONS DECIMAL

: TASK ;

7       LOAD    ( complex math words )
15      LOAD    ( vector words )
19      LOAD    ( fft words )
30      LOAD    ( demo fft words )
```

Screen # 7
```
( square and square root )
( from Forth Dimensions Vol. IV #1, pg 9-10 by Klaxon Suralis )
: D2*       2DUP D+ ;

: EASY-BITS
  0 DO  >R D2* D2* R@ - DUP 0<
        IF   R@ + R> 2* 1-
        ELSE      R> 2* 3 +
  THEN LOOP ;

: 2'S-BIT
  >R D2*  DUP 0<
  IF   D2* R@ -  R> 1+  ELSE D2* R@ 2DUP  U<
       IF   DROP R> 1-  ELSE  - R> 1+ THEN
  THEN ;
-->
```

Screen # 8
```
( square and square root, cont. )

: 1'S-BIT
  >R DUP 0<
  IF   2DROP R> 1+
  ELSE D2*  32768 R@ DU< 0= R> +
  THEN ;

: SQRT          ( d --- s )
  0 1 8 EASY-BITS  ROT DROP 6 EASY-BITS
  2'S-BIT  1'S-BIT ;

: SQR       DUP M* ;          ( s --- d )
-->
```

Screen # 9
```
( complex data type and operations )

: X@        2@ ;                ( addr  --- x )
: X!        2! ;                ( x  addr --- )

: XVARIABLE             ( --- )
  CREATE    WSIZE 2* HERE OVER  ( two elements per complex no.)
            ERASE ALLOT         ( initialized to zero )
  DOES> ;

: XCONSTANT             ( x --- )
  CREATE    HERE WSIZE 2* 2DUP      ( two elements per cmplx )
            ERASE ALLOT  X!         ( save TOS into addr )
  DOES>     X@ ;                    ( return complex constant)
-->
```

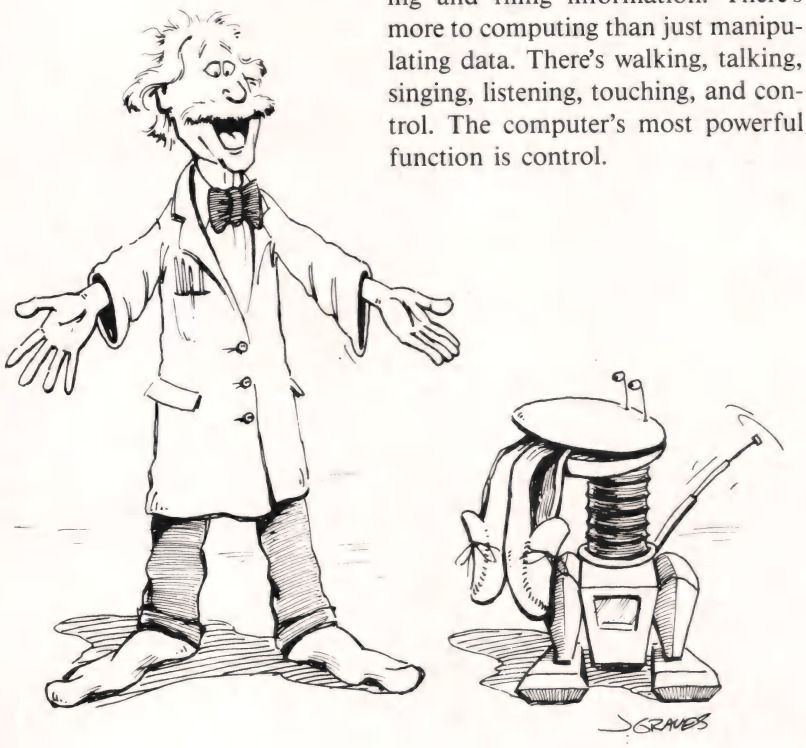
(Continued on page 44)

ROBOTICS AGE

A Real-Time Experience

Learning how to program was the easy part. Now, what are you going to do with your knowledge? How about writing a check balancing program? Maybe you could start developing that wonderful data base to keep track of everyone's birthdays. However, there are more interesting challenges.

Computers are very useful for tracking and filing information. There's more to computing than just manipulating data. There's walking, talking, singing, listening, touching, and control. The computer's most powerful function is control.



Real-time computer techniques can control factories and machinery, monitor your home environment and protect it from intruders, and operate little mechanical friends which will take out the garbage.

The near future will show us machines which respond to human voice controls, are capable of finding their own way around a house or factory floor, and are able to make their own decisions. *Robotics Age* teaches you to design and work with the practical real-time applications of state-of-the-art microcomputer technology. Robots are simply machines which respond to their environments and can act on their own. Computers make these machines possible.

After all, many people claim their computers are user friendly—but how can your computer be user friendly if it doesn't come when it's called?

It's time for you to experience *Robotics Age.* Explore the frontiers of microprocessor applications. Use the subscription form below to start the flow of vital technical information you need.

YES! ***Sign me up TODAY for my personal subscription to Robotics Age, The Journal of Intelligent Machines.***

DD3

US Subscriptions

☐ 12 issues	$24
☐ 24 issues	$45
☐ 36 issues	$63

Canada & Mexico

☐ 12 issues	$28
☐ 24 issues	$53
☐ 36 issues	$75

Foreign

☐ 12 issues (surface)	$32
☐ 12 issues (Air Mail)	$68
☐ 24 issues (surface)	$61
☐ 24 issues (Air Mail)	$133
☐ 36 issues (surface)	$87
☐ 36 issues (Air Mail)	$195

Non US Subscription Rates:
Payable in US funds, drawn on a US bank. Subscription length will be adjusted downward on a pro-rata basis for any currency conversion charges. Foreign subscription orders may be paid in US dollars via MasterCard or VISA.

MasterCard VISA

RETURN WITH PAYMENT TO:
Robotics Age, Box 358
Peterborough, NH 03458

Name _____

Company _____

Address _____

Town _____

State _____ Zip/Postal Code _____ Country _____

☐ Bill Me

Credit Card Information

☐ MasterCard _____ Card Number

☐ VISA _____ Expiration Date

_____ Signature

Total amount Enclosed or Charged $ _____

Circle **no. 72** on reader service card.

Fourier Transform Listing (Listing Continued, text begins on page 34)

Screen # 10
```
( complex stack words )

: XDUP      2DUP ;              ( x  ---  x  x )
: XSWAP     2SWAP ;             ( x1  x2  ---  x2  x1 )
: XDROP     2DROP ;             ( x  --- )
: XOVER     2OVER ;             ( x1 x2 --- x1 x2 x1 )
: X2DUP     2OVER 2OVER ;       ( x1  x2  ---  x1  x2  x1  x2 )
-->
```

Screen # 11
```
( complex add, subtract, magnitude )

: X+        ROT + >R + R> ;             ( x1  x2  ---  x1+x2 )
: X-        ROT SWAP - >R - R> ;        ( x1  x2  ---  x1-x2 )
: X2/       2/ SWAP 2/ SWAP ;           ( x --- x/2 )
: |X|^2     SQR ROT SQR D+ ;            ( x --- mag_x**2 )
: |X|       |X|^2 SQRT ;                ( x --- mag_x )
: X'        SWAP NEGATE SWAP ;          ( x --- x_conjugate )
: X0=       0= SWAP 0= AND ;            ( x --- t/f )
-->
```

Screen # 12
```
( complex multiply )
-->

: X*        >R  X2DUP           ( x1  x2  n ---  x1*x2/n )
            ROT R@ */                 ( real part = )
            ROT ROT R@ */  -          ( re1*re2 - im1*im2 )
            R> SWAP >R >R             ( save partial result)
            ROT ROT R@ */             ( re2*im1 )
            ROT ROT R> */  +          ( + re1*im2 )
            R> ;                      ( imag real --- )
```

This word is replaced by one on the screen that follows. The high level FORTH definition here is for documentation purposes.

Screen # 13
```
( assembly def for complex multiply )

ASM     ( load assembler if not already loaded. )
CODE X*
    (SP)+ D0 MOVE,  (SP) D1 MOVE,  4 d(SP) D1 MULS,
    2 d(SP) D2 MOVE,  6 d(SP) D2 MULS,  D2 D1 L. SUB,
    D0 D1 DIVS,  (SP) D2 MOVE,  6 d(SP) D2 MULS,
    2 d(SP) D3 MOVE,  4 d(SP) D3 MULS,  D3 D2 L. ADD,
    D0 D2 DIVS,  8 # SP ADDQ,  D2 -(SP) MOVE,  D1 -(SP) MOVE,
    NEXT, END-CODE

-->
```
Stack indexing is used to keep the original data available as the complex product is built. When the product is complete, the original data is popped off the stack and the product pushed onto it.

Screen # 14
```
( more complex operations )

: X*!       OVER >R >R          ( ^x1 ^x2 n --- x2 (-- x2*x1/n)
            X@  ROT X@  R>              ( form x1*x2/n )
            X*   R> X! ;                ( save it into x2^ )

: X/        >R XSWAP XOVER      ( x1 x2  n  ---  n*x1/x2 )
            X' R@ X*                    ( x1*x2' )
            XSWAP |X| DUP R@ */         ( sqr|x2| )
            SWAP OVER  R@ SWAP */       ( real part )
            ROT ROT R> SWAP */ SWAP ;   ( imag part )
```

Screen # 15
```
( Vector definition )

: VECTOR        ( order scale word_size --- )
  CREATE    ROT 2DUP , ,          ( store order, then wsize )
            ROT ,                 ( then store scale )
            * DUP ,               ( bytes of entry storage )
            HERE OVER ERASE       ( erase vector )
            ALLOT                 ( allot dictionary for entries)
  DOES> ;                         ( return address of base )

-->
```

This data structure holds a vector that contains "order" entries. Legal indicies are in the range of 1 to "order". "Scale" is used in numeric operations on entries, and the "word_size" field describes the number of bytes per entry.

Screen # 16
```
( vector operations )

: &ORDER        ( vector_descriptor --- order_of_vector )
  @ ;

: &ESIZE        ( vec_desc --- size_of_entries )
  WSIZE + @ ;

: &SCALE        ( vec_desc --- scale_of_vec )
  WSIZE 2* + @ ;

: &SIZE         ( vec_desc --- size_of_data_area )
  WSIZE DUP 2* + + @ ;
-->
```

(Continued on page 46)

Fourier Transform Listing (Listing Continued, text begins on page 34)

Screen # 17

```
( matrix operations, cont. )
-->

: [I]          ( index  vec_desc --- addr_of_entry )
  DUP >R &ORDER                ( save vec_desc, get entrysize)
  OVER < OVER 1 < OR ABORT" Vector bounds exceeded... [I]"
  1-  R@ &ESIZE *              ( index * entrysize )
  WSIZE 4 * + R> + ;           ( +offset to entries )
```

This word calculates a storage address given a column number and a matrix descriptor. If the desired address is outside of the alloted vector storage area, an exception is generated. This word is usually followed by "@" or "!". The following screen contains the assembly language equivalent.

Screen # 18

```
( assembly language version of [i] )

ASM
CODE [I]                ( index vec_desc --- addr )
    (SP)+ 0S MOVE,  0 d(BP, 0S) A0 LEA,          ( a0=vec addr)
    (SP)+ D0 MOVE,  1 # D0 CMPI,  1$ BLT,        ( index<1 )
    (A0) D0 CMP,    1$ BGT,                      ( index>max)
    1 # D0 SUBQ,  2 d(A0) D0 MULS,  D0 0S ADD,   ( word size)
    0 # 0S ADDQ,  0S -(SP) MOVE,  NEXT,          ( success )

1$: -1 # -(SP) MOVE, 2$ # 0S MOVE,               ( force abort")
    0 d(BP, 0S) IP LEA,  NEXT,

2$: ] ABORT" Vector bounds exceeded... [I]" [
END-CODE
```

Screen # 19

```
( complex vector defs )

: XVECTOR       ( order scale --- )
  WSIZE 2* VECTOR ;      ( declare a vector with two words/ent.)

: X@V           ( index vec_desc --- x )
  [I] X@ ;               ( fetch vector entry pointed by index)

: X!V           ( x index vec_desc --- )
  [I] X! ;               ( store into vector at position index)
-->
```

The constant WSIZE (on line 3) returns the word size of the Forth system used. For a 16-bit implementation (like LMI Forth) it is 2, for a 32-bit system (like LMI Forth+) it is 4.

Screen # 20

```
( FFT constants and variables )

10000   CONSTANT        10K             ( scaling constant )
        VARIABLE        DIR             ( direction of fft )
        VARIABLE        VEC             ( current vector base)
        VARIABLE        N               ( points in fft )
        XVARIABLE       U               ( cmplx angle counter)
        XVARIABLE       W               ( cmplx angle incr)
-->
```

Screen # 21

```
( bit reversal of vector )

: V[I][J]               ( i  j  --- [i]  [j] )
  VEC @ [I] SWAP                ( calculate first addr )
  VEC @ [I] SWAP ;              ( then second addr )

: XSWAPV                ( i  j  --- )
  V[I][J]                       ( calculate addresses )
  2DUP SWAP >R >R               ( save one copy on ret stack)
  X@ ROT X@  R> X! R> X! ;      ( swap cmplx numbers )
-->
```

Screen # 22

```
( bit reversal, cont. )

: BIT-REVERSE           ( --- )
  N @ 1 SWAP 1 DO               ( for i=1 to size of vector )
    DUP I > IF                  ( compare indicies )
      DUP I XSWAPV              ( swap vector entries )
    THEN
    N @ 2/ SWAP                 ( let incr=points/2 )
    BEGIN   2DUP <              ( while incr < indx )
    WHILE   OVER - SWAP 2/ SWAP ( indx=indx-incr, incr=incr/2)
    REPEAT  +                   ( indx=indx+incr )
  LOOP DROP ;                   ( drop indx )
-->
```

Screen # 23

```
( butterfly calculation )

: BFLY          ( i  j  --- )
  V[I][J] 2DUP >R >R            ( calc actual addresses )
  X@  U X@  10K X*              ( form temp product )
  ROT X@  XSWAP X2DUP           ( prepare stack for + and - )
  X- >R >R  X+ R> R>            ( Fi-temp, Fi+temp )
  DIR @                         ( test for fwd/inv transform )
  IF   XSWAP                    ( IFFT, no divide )
  ELSE X2/ XSWAP X2/            ( FFT, divide by points )
  THEN R> X!  R> X!  ;          ( store products )
-->
```

Screen # 24

```
( 2** and log2 )

: 2**           ( n  ---  2**n )
  0 MAX WSIZE 8 * 1- MIN        ( limit input data )
  1 SWAP 0 ?DO 2* LOOP ;        ( DON'T loop if n=0 )

: LOG2          ( n  ---  log2[n] )
  DUP 0= ABORT" Can't take log of zero. "
  0 SWAP                        ( limit one word integer )
  BEGIN   DUP 0>                ( test upper bit )
  WHILE   2* SWAP 1+ SWAP       ( if zero, shift and incr count)
  REPEAT  DROP                  ( done, drop remainder )
  WSIZE 8 * 1- SWAP - ;         ( return log base 2 of input)
-->
```

(Continued on page 48)

Fourier Transform Listing (Listing Continued, text begins on page 34)

Screen # 25
```
( table of cosines and sines for 2**0 to 2**7 )

VARIABLE WTAB   8 WTAB !        ( dummy vector entry )
        4 , 10000 , 32 ,        ( entries, wsize, scale, data)
       -10000 ,  00000 ,  00000 , -10000 ,  07071 , -07071 ,
        09239 , -03827 ,  09808 , -01951 ,  09952 , -00980 ,
        09988 , -00491 ,  09997 , -00245 ,
-->
```

This is a table of cosines and sines for angles of pi/n where n is an integral power of two. The first entry is pi/1, cos is -1 and sin is 0. Next is pi/2, cos is 0 and sin is 1. Entries up to pi/128 currently, but it can be extended for larger FFT's.

Screen # 26
```
( inner-loop )

: INNER-LOOP    ( startpt increment --- )
  DUP 2/                        ( increment/2 is bfly offset)
  ROT N @ 1+ SWAP               ( limits are startpt...N )
  DO    I 2DUP +                ( do a butterfly computation )
        BFLY OVER               ( between I and I+offset )
  +LOOP 2DROP ;                 ( increment loop )
-->
```

Screen # 27
```
( fft-kernel )

: FFT-KERNEL            ( --- ) ( N, VEC, must be set )
  N @ LOG2 1+ 1                 ( do for i=0 to log2[N] )
  DO  0 10K U X!                ( init U to 1+j0 )
        I WTAB X@V              ( get 1/-arg[pi/I])
        DIR @ IF X' THEN W X!   ( take conjugate if IFFT )
        I 2** DUP 2/ 1+ 1       ( increment and offset counter)
        DO   I OVER INNER-LOOP  ( figure innermost loop )
             W U 10K X*!        ( set new value for angle )
        LOOP DROP               ( drop increment value )
  LOOP ;

-->
```

Screen # 28
```
( fft and ifft )

: INIT          ( vector --- )
  DUP VEC !
  &ORDER N ! ;                  ( initialize size variable)

: DO-VECTOR     ( vector --- )
  INIT                          ( initialize size field )
  BIT-REVERSE                   ( bit swap input )
  FFT-KERNEL ;                  ( perform butterflies )

: FFT         0 DIR ! DO-VECTOR ;
: IFFT       -1 DIR ! DO-VECTOR ;
```

Screen # 29
```
( spare screen )
```

Screen # 30
```
( plotting words for complex vector )

VARIABLE MAG      10K MAG !     ( largest value in vector )
VARIABLE MIDDLE    35 MIDDLE !  ( zero position, in char units)
VARIABLE PLOTBUF   68 ALLOT     ( buffer for building image )

: MAP                   ( n --- n )
  MIDDLE @ -
  35 MAG @ */ ;                 ( map into character coord. )
-->
```

Screen # 31
```
( put char into buffer before printing line )

: BUF!                  ( char n --- )
  69 MIN 0 MAX                  ( limit access to buffer )
  PLOTBUF SWAP ROT FILL ;       ( store character into buffer)

: ?BUF!                 ( char n --- )
  DUP 69 <= OVER 0 >= AND       ( check for out-of-bounds )
  IF   PLOTBUF + C!             ( if in bounds, print char )
  ELSE 2DROP
  THEN ;
-->
```

Screen # 32
```
( plot one line )

: PLOTDAT               ( n --- )
  PLOTBUF 70 BLANK              ( blank out plotter buffer )
  ASCII : 0 MAP ?BUF!           ( "zero" line if in range )
  ASCII * SWAP MAP BUF!         ( plot character for data )
  PLOTBUF 70 -TRAILING TYPE ;   ( print the plot buffer )

: PLOTINDX              ( n --- )
  2 SPACES 5 .R                 ( print index number )
  2 SPACES ASCII ¦ EMIT ;       ( and right-hand bar )
-->
```

Screen # 33

```
( plot magnitude for entire vector )

: PLOTMAG                ( vec_desc --- )
  CR  DUP &ORDER 1+ 1          ( do for each entry in vector)
  DO  I PLOTINDX               ( get entry )
      I OVER X@V               ( get entry )
      !X! PLOTDAT CR           ( plot magnitude value )
  LOOP CR DROP ;               ( drop vec. descriptor )

-->
```

Screen # 34

```
( load data from stack )
( TOS is vector descriptor, next is number of points, then )
( scale factor, data points follow )

: VECTOR!
  OVER >R                      ( save number of data points )
  SWAP OVER !                  ( reset size of vector )
  SWAP OVER WSIZE 2* + !       ( reset scale of vector entries)
  1 R> DO                      ( for each vector entry, )
     SWAP OVER  0 ROT
     I  4 ROLL X!V
  -1 +LOOP DROP ;              ( work backwards thru vector )
-->
```

Screen # 35

```
( data for IBM over 1983, by week )

100 64 XVECTOR IBM
9663    9913    9463    9738    9738    9638    9863    10038
10225   10075   9988    10213   10163   10388   11013   11725
11700   11763   11650   11963   11300   11400   11425   12113
12300   12100   12150   12013   12438   12038   11975   11850
12250   11783   11975   12225   12313   12663   12688   13225
13175   12700   12800   12225   12688   12350   12100   11788
12225   12088   12363   12200   0       0       0       0
0       0       0       0       0       0       0       0
100 64 IBM VECTOR!

( This data is the closing price of IBM common stock on Friday
  of each week for 1983.  Eights are converted to decimal frac-
  tions.  The data is "padded" to 64 elements for the FFT. )
```

End Listing

Computing with Streams

by L. L. Odette

How do you work with an infinite list in finite storage? You use lazy evaluation, one of the concepts the author of this theoretical piece explores via Forth words.

The attraction of Forth is that you can do almost anything you want with it. One of the things I like to do is build word sets to explore programming issues. In this article, I'll present Forth words for experiments with data-flow-oriented computation, based on a data structure called a stream.

Much current interest centers on data-flow programs and machine organizations because of the advantages to be gained from concentrating more on the flow of data in a computation than on the sequence of steps.[1,2] The advantages include a nicer mathematical structure for the program, increased program modularity, and a program organization that facilitates use of parallelism. Besides any other advantages, organizing a program in terms of data flow can be a natural way to solve a problem that can be expressed in signal processing terms.

One way to make the data-flow organization of a computation concrete is to implement a data structure called a stream. Think of a stream as a list—nothing more than a sequence of data objects representing the data flow during a computation. The computation is organized as a composition of processes that take streams as input and produce other streams as output.

The mathematical niceties of stream processing arise from the lack of variables and associated concepts of "state." Computations organized about objects with coupled state variables (von Neumann style[3]) are difficult to make modular and are susceptible to bugs due to side effects of steps in the computation. All that happens in a pure stream computation is that one stream becomes some other stream, so you program in a functional programming style. Pure functional programming prohibits anything that changes the values of variables; it thereby deals with side effects by ruling them out altogether. Functional programming has been promoted as an alternative to von Neumann-style programming[3,4].

Stream computations can also be highly modular. This is because most of the processing procedures can be abstracted to a small number of higher-order procedures. For example, there are stream *filters*, that remove objects from the stream if they satisfy a given condition. There are stream *maps* that apply a given function to each object in the stream. There are also stream *accumulate* operations that combine stream objects using some procedure. Programming with streams is largely a matter of designing the proper composition of filters, maps, and accumulations.

Parallelism can be exploited because processes communicate only through the streams. There is no shared memory, and each reference to the stream is identical, so processes that are computationally parallel could be executed by different machines.

My goal is to illustrate how stream processing can be implemented in Forth, to provide a starting point for those who might experiment with these ideas. The implementation introduces other interesting topics, such as delayed evaluation and demand-driven program execution, which are also part of current computing research areas[1].

L.L. Odette, Boston University, College of Engineering, 110 Cummington St., Boston, MA 02215.

Streams in Forth

So, how can streams be implemented in Forth? Well, the straightforward implementation of streams as lists would not allow the full exploitation of the power of stream processing. If implemented properly, stream processing techniques allow us to model communicating sequential processes, implement mutual recursion, and process potentially infinite lists (as long as we process only finite parts of those lists).

If we are going to process an infinite list, the list can't be computed before it is processed, and some form of delayed evaluation is necessary. Delayed evaluation in the Forth context means that the inner interpreter is redirected so that a Forth word executes a little and is then delayed while other Forth words execute. Program execution proceeds as a quasi-parallel interweaving of words—a natural interpretation of mutually recursive definitions.

Demand-driven execution is associated with delayed evaluation, because the delayed part of a word must be explicitly forced to execute. The only way a result is produced during the computation is by constantly forcing execution of delayed words—a mode of execution sometimes called lazy evaluation. Demand-driven execution occurs when instructions are executed only when the result they produce is needed by another, already selected instruction.

These, then, are the basic ingredients for data-flow processing: a data structure representing the flow of data in a computation, delayed evaluation to suspend computations that produce the elements, and demand-driven execution to force data to be made available when required.

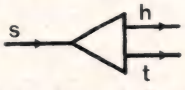
Before discussing the details of the Forth implementation, I'll set the stage with some basic operations and stream processing examples, introducing some useful notation in the process.

Basic Operations and Notation

Constructors and Selectors

There is a single primitive process that is needed to build streams: the stream constructor. The stream constructor will be represented by a triangle symbol, as follows:

This symbol represents the process of constructing a stream *s* by taking a data object *h* and tacking it onto the beginning of the sequence of data objects *t*. Because streams represent sequences of elements, *h* is the head (first element) of the new sequence *s*, and *t* is the tail (rest) of the sequence. The data flow direction is indicated by arrows; this allows the triangle symbol to do double duty as a representation both of the inverse of the constructor, the stream selector. The representation of the selector is



Note again that the direction of the arrows indicates the direction of data flow. The selector operator takes a stream *s* and makes available the head of the stream *h* and the tail of the stream *t*. In my Forth implementation of stream processes I define the selector words HEAD and TAIL, which expect a stream on the stack, and then leave the head and the tail of the stream, respectively.

It is important to note that while a stream represents a list, the only way to access elements in the list is to build a composition of TAIL operations followed by HEAD. For example, the fourth element of a stream is obtained by

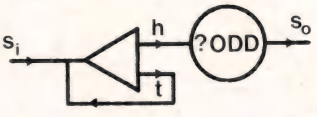
```
TAIL TAIL TAIL HEAD
```

The first TAIL takes a stream and leaves a new stream whose head is the second element of the original stream. The next TAIL leaves a stream whose head is the third element of the original stream. The final TAIL leaves a stream whose head is the fourth element of the original stream and is selected by HEAD.

The selectors and constructors are the only primitive operators on streams. You build higher-order stream processes from them by adding operations on the data elements of the stream. Filters, maps, and accumulations are some of the more common higher-order processes for operating on streams and demonstrate the high level of abstraction possible in stream processing.

Filters, Maps and Accumulations

One common stream process is the *filter*. A filter just takes a stream and returns a new stream by removing from the input stream all the elements satisfying a specified condition. For example, supposing that the word ?ODD tests a number for oddness and returns a flag indicating the result of the test, then the following represents a process that filters out the odd elements of an input stream of numbers.



In the process above, the selector picks out the head of the input stream and ?ODD tests it; then it is passed through to the output if it is even. The tail of the input stream is fed back and the process repeats, selecting the head of the tail, testing for oddness, feeding back the tail of the tail, and so on.

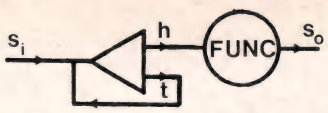
Filters are common elements in data-flow computations; this suggests that an abstraction of the filter above to a generic filter would be useful in designing programs. To build a particular filter from the generic filter, you must combine a predicate (filter word) with the generic filter process. Forth filter definitions would be of the form:

```
: name predicate FILTER ;
```

where *name* is the name given to the filter, and *predicate* represents any word that applies a test to the head of the input stream (leaving true or false on the parameter stack as does the word ?ODD, above). FILTER is the generic filter word, perhaps implemented as an immediate word that sets up a filter at compile time, using the code address compiled for predicate. At runtime the particular filter word expects an input stream on the stack and produces an output stream, filtered as specified by predicate.

A process somewhat similar to a filter is a *map*, a term that takes its name from LISP. What a map does is apply a func-

tion to each element of a stream to produce a new stream. Using the triangle notation, a map process looks like this:



As in the filter process, the selector obtains the head of the input stream s_i and then feeds back the tail. The function FUNC is applied to each element of the input stream to produce the output stream s_o. For example, if the function were one that doubled each element, an input stream 1, 2, 3, 4, . . . would be transformed to the output stream 2 , 4, 6, 8, ... Like filters, maps can be abstracted to an operation combining a function and a generic map. Forth definitions for particular maps would be of the form

: name function MAP ;

where *name* is the name of the MAP process and *function* represents the function word to be applied to each element in the input stream in order to produce the output stream. MAP is the generic map process.

A somewhat more complicated stream process is an *accumulation.* An accumulate process builds an output stream by applying a binary operator to successive elements of the input and output sequences and accumulating the results. Specification of an accumulate process requires the accumulate operator and an initial condition. The head of the output stream is given by the initial condition. The output stream is then fed back to be combined, via the specified operator, with the input stream. The results form the tail of the output stream s_o.

For example, an accumulate based on the + operator would return a stream of partial sums if given a stream of numbers. The partial sum process, based on + with initial condition 0, is shown in Figure 1 (at right). The constructor Element A forms the output stream, starting with 0. Selector Element B picks out elements of the input integer stream in sequence, and Selector C picks out sequential elements of the output stream. Passing the selected elements through + provides the tail for Constructor A by adding elements of the input sequence to corresponding elements of the partial sum sequence.

Filters, maps, and accumulations are some of the basic building blocks for organizing data-flow computations, and using them makes it relatively easy to understand what a stream processing program is supposed to do. There are suggestions that a data-flow analysis of any computation may be couched largely in terms of these operations, as Waters'[5] analysis of the Fortran programs in the IBM Scientific Subroutine Package showed that nearly 60% of the code fit a stream processing description involving filters, maps, and accumulations.

A Prime Example

For a more complicated program involving streams, consider how stream processing operations might be used to construct a list of all prime numbers. One way to generate the primes is to implement a network for the prime sieve of Eratosthenes and then feed it the sequence of integers starting with the integer 2. The output is the sequence of primes.

The sieve of Eratosthenes can be described in data-flow terms as follows:

1) The head of the input stream is prime. Call it *P*.
2) Filter all multiples of *P* from the tail of the input stream.
3) Feed the filtered stream back to (1).

The sequence of *P*'s generated in step 1 is the sequence of prime numbers, starting with the prime number 2. Part 2 of the sieve description is a stream filter that is slightly different from the filter described earlier in that it takes an input number *P* as a parameter and then uses DIVISIBLE-BY-P as the filter predicate. The DIVISIBLE-BY-P filter removes all integers that are divisible by the integer *P* from the input stream. Using a filter of this sort, the stream processor implementing steps 1 through 3 (the sieve) is shown in Figure 2 (below).

The sieve works as follows. In the stream processor diagrammed in Figure 2, the input stream is a sequence of integers 2, 3, 4, ... The selector removes the head of the input stream and sends it to the constructor, so that the head of the input stream becomes the head of the output stream. Thus 2 is the first prime generated by the sieve. This is step 1 in the description above. The selector also sends the head and tail of the input stream to the DIVISIBLE-BY filter. The head of the input stream is the input parameter to DIVISIBLE-BY, so that DIVISIBLE-BY produces the stream 3, 5, 7, 9, 11, 13, 15, . . . — i.e., the tail of the input stream with all multiples of 2 removed. This is step 2 in the description above. In step 3, the output of DIVISIBLE-BY is passed through a prime sieve process to produce the tail of the output stream.

The prime sieve that DIVISIBLE-BY feeds processes the stream 3, 5, 7, 9, 11, 13, 15, . . . as per the sieve description: the head of the input stream becomes the head of the output stream. Thus 3 is the second prime generated by the sieve. DIVISIBLE-BY removes all multiples of 3 from the tail of the input stream to produce the stream 5, 7, 11, 13, 17, . . . — i.e., the stream 2, 3, 4, 5, . . . with all multiples of 2 and 3 removed. The stream 5, 7, 11, 13, 17, . . . is then passed through a sieve to produce the tail of the output stream.

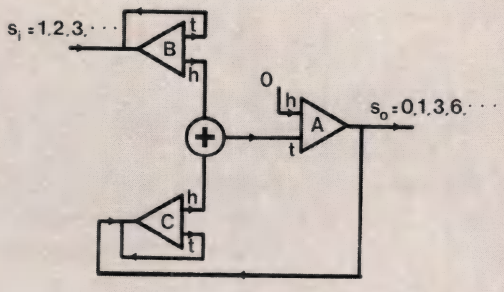


Figure 1

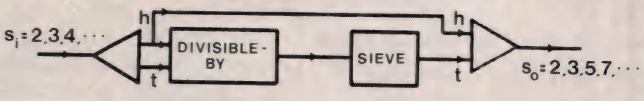


Figure 2

What is unusual about this network is that it contains a copy of itself. The input represents an infinite sequence of integers, as the output stream of primes also represents an infinite sequence, and this means that the above represents a program that is infinitely long. In implementations employing demand-driven execution, only as much of the stream as is necessary (demanded as output) is ever computed, and so only as much of the program as is needed exists. During execution, the program in effect builds itself and builds only as much as is required.

The Forth Implementation

The key to a useful implementation of streams and stream processing words is to delay computing anything until the result is needed by some other computation. This is demand-driven computation. To support this mode of execution, the data structure representing a stream will be considered as consisting of a recipe for computing the head and tail of the stream together with the ingredients required by the computation. In Forth, a stream on the stack will consist of n stack entries with the structure:

($n-2$ elements), pointer, n . . .

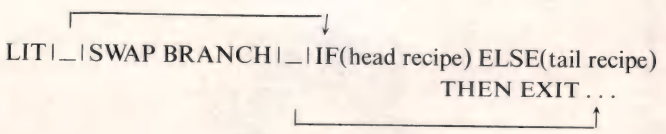
The top stack entry (n) indicates the total number of stack entries comprising the stream. The count is needed in order to manipulate these objects (i.e., we can expect to need words such as STREAM-SWAP, STREAM-DROP, and so on). The next component of the stream is a pointer to the recipe for computing the stream head or tail. The final $n-2$ stack entries comprise the basic ingredients used in the recipe.

The suggested implementation of Forth stream constructor words is given in screen 219 (page 58). The constructors are IMMEDIATE words and can only be used in COLON definitions. They have the form:

```
MAKE-HEAD (head recipe)
MAKE-TAIL (tail recipe)
END STREAM
```

where (head recipe), (tail recipe) represent sets of Forth words that take the ingredients and compute the head and tail, respectively.

At compile time, MAKE-HEAD (head recipe) MAKE-TAIL (tail recipe) END-STREAM is expanded to.



At runtime, MAKE-HEAD expects the parameter stack to contain the stream length and ingredients. The full stream is then constructed by putting a pointer to the IF clause on the stack and performing a SWAP. BRANCH delays any computation by transferring control beyond the IF clause. Note that the only effect MAKE-HEAD . . . MAKE-TAIL . . . END-STREAM has at this time is to construct a stream from the data on the stack. Execution of the recipes in the body of the

construct is delayed until they are needed to process the stream.

The recipes are forced to execute by the selector words HEAD and TAIL. These are implemented as

```
: HEAD DROP  >R 1 ;
: TAIL DROP  >R 0 ;
```

At runtime they take a stream on the stack, transfer the pointer to the IF (head recipe) ELSE (tail recipe) THEN clause from the parameter stack to the return stack and leave a 1 or 0 on the parameter stack. Control is thereby transferred to the appropriate part of the recipe. When the recipe is complete, control is transferred back to the word following the HEAD or TAIL selector.

A simple example of the use of the stream constructor is provided by the word INTEGERS-FROM, which takes an unsigned integer on the stack and leaves a stream representing all the integers greater than or equal to the input integer. You can then access elements of this (infinite!) list using HEAD and TAIL. INTEGERS-FROM is defined as

```
: INTEGERS-FROM 3 MAKE-HEAD
                  MAKE-TAIL 1 + MYSELF
                  END-STREAM ;
```

where MYSELF is Levan's[6] recursion word.

When executed, INTEGERS-FROM leaves a three-element data structure on the stack—the top element being the count, the next element a pointer, and the third element the starting integer. The recipe for computing the head of the stream is NOP, because the head of the sequence is all that is left on the stack by HEAD. The recipe for computing the tail is to take the head of the sequence, add one, and then form a new stream by executing INTEGERS-FROM.

INTEGERS-FROM represents an infinite sequence in the usual mathematical sense that we never run out of elements of the sequence. The finite word size of stack entries (say, 16 bits) means that in practice INTEGERS-FROM repeats itself every 2^{16} elements, so that, as defined above, it is more accurate to say that INTEGERS-FROM is a representation of all integers greater than or equal to the input integer, MOD 2^{16}.

(MAKE-HEAD(head recipe) MAKE-TAIL(tail recipe) END-STREAM), HEAD and TAIL are the basic stream constructor and selector words. Implementations of stream filters, maps, and accumulations are given in screens 223 – 225 (pages 60 – 61). Other words useful for stream manipulation are given in screens 220 – 222 (pages 58 – 60).

Yet Another Forth Prime Sieve

The sieve of Eratosthenes, implemented with Forth stream processing words, is given in screen 226 (page 61). The basic element of the sieve is DIVISIBLE-BY, a stream filter that removes stream elements that are divisible by some number. DIVISIBLE-BY expects a number and a stream on the stack, filters the stream, and then packages the result in a new stream. The recipe for computing the head of the filtered stream is to drop the top stack entry (the divisor) and execute HEAD. The recipe for computing the tail is

```
>R TAIL R> MYSELF
```

that is, save the divisor, find the tail of the input stream, and filter the tail with DIVISIBLE-BY.

SIEVE processes the input stream according to the following recipes: The head of the output stream is the head of the input stream. The tail of the output stream is found by first filtering the tail of the input stream through DIVISIBLE-BY, using the head of the input stream as the divisor. This filtered stream is then passed through a copy of SIEVE. The recipes follow the three steps in the description of the sieve of Eratosthenes given earlier.

Finding the *n*th prime is straightforward:

```
: PRIMES 2 INTEGERS-FROM SIEVE ;
   (make a stream of all primes)
: SELECT 0 DO TAIL LOOP HEAD . ;
   (select an element of a stream)
PRIMES 10 SELECT.
   (select the 10th prime)
```

There are several noteworthy things about this program. In a sense, the program works back from the request to get the tenth prime, using as much memory as is necessary to compute what has been requested. The larger the prime requested, the greater the memory requirements for the computation.

Because the stream is on the stack, the available stack space imposes a practical limit on how many primes can be computed via this implementation of the sieve. For example, after 25 primes have been computed, the tail of the output stream is 80 stack elements, representing the input stream 2, 3, 4, . . . with all multiples of the first 25 primes removed. The 80 stack elements, starting from the stack bottom, are three elements specifying the original stream (INTEGERS-FROM), three elements for each invocation of DIVISIBLE-BY (count, recipe pointer, and division) and two elements of the sieve stream header (count, recipe pointer).

Note also the mutual recursion implicit in the fact that DIVISIBLE-BY processes streams that are constructed by SIEVE, while SIEVE processes streams constructed by DIVISIBLE-BY.

A Final Example

Writing programs to experiment with new ways of writing programs could be called metaprogramming. I've presented these Forth words for stream processing as an exercise in metaprogramming, hoping that those who like to experiment in Forth will improve and extend them and that those who solve practical problems with Forth may recognize where similar techniques could be useful in their own programming tasks.

For an application of stream processing on the practical side, I tried designing Forth words to implement a Forth interpreter (screens 227 – 236, page 61 – 67). The programming issues in this task involve deciding how the problem should be posed to fit the data-flow model and choosing a structure for the data. I chose to make the output a stream of environments, each environment consisting of a representation of the parameter and return stacks, plus a pointer into the parameter field of the word being interpreted. The input is an initial value for the environment stream, specifying the

contents of data and return stacks and where to start the interpreter. Environment formats are discussed in the legend to Figure 3 (page 56), which illustrates the data-flow structure of the program using triangle notation.

The program is built from three maps and an accumulation. The maps, with their input and output streams, are

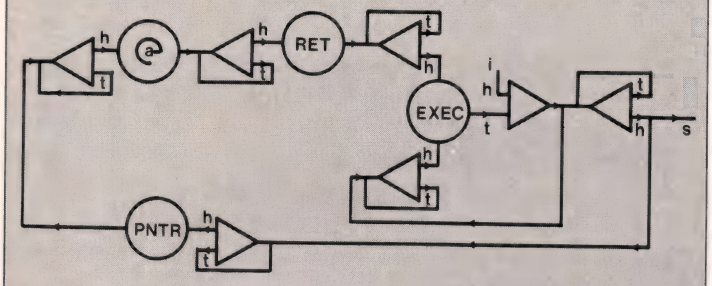
MAP	INPUT-STREAM	OUTPUT-STREAM
PNTR	Environments	Parameter field addresses
@	Parameter field addresses	Contents of parameter field addresses(code field addresses)
RET	Code field addresses	Modified code field addresses

The result of applying these three maps in sequence is a three-step process. The first step takes an environment and drops the stack representations, leaving the parameter field pointer (PNTR). Next, the content of that address is obtained (@), yielding a code field address. The stream of code field addresses is then mapped to a modified set of code field addesses (RET) according to the following rule: If the domain word (input to RET) affects the inner interpreter instruction register (LIT, BRANCH, 0BRANCH, etc.) or modifies the return stack (<DO>, <LOOP>, etc.), then the image of the map is the code address of a word that has an effect on the environment data object that is congruent to the effect of the domain word on the instruction register or return stack. Otherwise, the image code address is identical to the input code address. Screens 229 and 230 contain code for image words of some, but not all, of the words that alter the instruction register or return stack.

The resulting stream of code addresses is fed to an accumulation based on a modified EXECUTE operation.

The result is a program that you can use to single-step through a colon definition, displaying the instruction pointer and the contents of the data and return stacks. Lines 9 – 10 and 13 – 14 of screen 236 show how to prepare a stream to do this. The word STEPS defined in screen 236 expects two environment streams on the parameter stack and will single-step through the two colon definitions in parallel. Extension to $n > 2$ words in parallel is left as an exercise. You may wish to experiment with parallel execution of words that communicate through some shared variable. Other worthwhile exercises include translating the words of screens 220 – 221 to machine code. This will significantly increase speed.

The point of this example is simply that a certain conceptual clarity results from organizing a program so that the data-flow structure of the problem is manifest in the procedures. The description of the operation of the interpreter as I've given it—a sequence of parameter field addresses is generated, the contents of parameter field addresses are used to generate a sequence of code field addresses, and the sequence of code field addresses are executed—has parts that are clearly separable and identifiable in the stream implementation. It follows that the network representation of the data flow given in Figure 3 leads fairly directly to a program. Metaprogramming involves thinking about how direct the transformation is and how useful the result.



Each stack is represented by the stack elements and a count. The count is the number of stack elements plus 1, so stack representations can be manipulated by the stream manipulation words STREAM-DUP, STREAM-SWAP, etc. An empty stack is therefore represented by the count, 1. An environment consists of a return stack representation, followed by a parameter stack representation; it is topped with a pointer into the parameter field of the word being interpreted.

The accumulation based on EXEC takes an environment (the output stream) and a code address (input stream) and forms a new environment. The *i* in the figure represents the initial environment for the accumulation.

Figure 3

Data Flow Organization of Forth Inner Interpreter.

Acknowledgements

I would like to thank John Kalafatas of Advanced Micro Devices for his support and for arranging a loan of computer equipment, which greatly facilitated this work.

References

1. Treleaven, P. C., Brownbridge, O. R., and Hopkins R. P., 1982. "Data-driven and demand-driven computer architecture." *ACM Computing Surveys* 14 (1) : 93 – 143.
2. Lerner, E. J., 1984. "Data flow architecture." *IEEE Spectrum* 21 (4) : 57 – 64.
3. Backus, J., 1978. "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs." *Comm. ACM* 21 (8) : 613 – 641.
4. Henderson, P., 1980. *Functional programming.* Prentice-Hall International, Inc., London.
5. Waters, R. C., 1979. "A method for analysing loop programs." *IEEE Trans. on Software Engineering* 5 (3) : 237 – 247.
6. LeVan, J., 1980. "Recursion—the eight queens problem." *Forth Dimensions* 2:6.

DDJ

A Professional Quality Z80/8080 Disassembler

REVAS Version 3

Uses either ZILOG or 8080 mnemonics
Includes UNDOCUMENTED Z80 opcodes
Handles both BYTE (DB) & WORD (DW) data
Disassembles object code up to 64k long!
Lets you insert COMMENTS in the disassembly!

A powerful command set gives you:

INTERACTIVE disassembly
Command Strings & Macros
On-line Help
Calculations in ANY Number Base!
Flexible file and I/O control
All the functions of REVAS V2.5

REVAS:

Is fully supported with low cost user updates
Runs in a Z80 CPU under CP/M*
Is normally supplied on SSSD 8" diskette
Revas V 3...$90.00 Manual only...$15.00
California Residents add 6½% sales tax

REVASCO
6032 Charlton Ave., Los Angeles, CA. 90056
(213) 649-3575

*CP/M is a Trademark of Digital Resaerch, Inc.

Circle **no. 70** on reader service card.

SMALL C FOR IBM-PC

Small-C Compiler Version 2.1 for PC-DOS/MS-DOS
Source Code included for Compiler & Library
New 8086 optimizations
Rich I/O & Standard Library

$40

CBUG SOURCE LEVEL DEBUGGER FOR SMALL C

Break, Trace, and Change variables all on the source level
Source code included

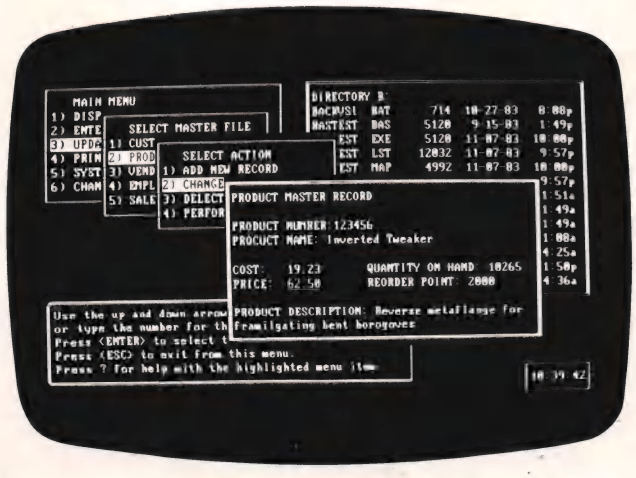
$40

Datalight
11557 8th Ave. N.E.
Seattle, Washington 98125
(206) 367-1803

ASM or MASM is required with compiler.
Include disk size (160k/320k), and DOS version with order.
VISA & MasterCard accepted. Include card no. & expiration date.
Washington state residents include 7.9% sales tax.
IBM-PC & PC-DOS are trademarks of International Business Machines
MS-DOS is a trademark of Microsoft Corporation.

Circle **no. 21** on reader service card.

WE DO WINDOWS.



Don't just put your applications in windows—put windows in your applications with VSI—the window manager.

VSI is a high-speed screen management tool. You can create up to 255 simultaneously active overlapping windows—large or small—for any application program. Read to or write from any window and display them with borders and user declared priorities. VSI is callable from any compiled language and supports all color and monochrome video attributes.

Cut Development Time

VSI's powerful primitives simplify your screen management chores with a complete library of functions. And you can preview and edit your screen layout before you actually program it.

But that's only the beginning.

Free Demo Disk

Our free hands-on demo disk will have you doing windows, too. Return the coupon with $4.50 for postage and handling.

MasterCard or Visa accepted with phone orders only.

VSI is used with IBM PC, XT and compatibles as well as TI Professional, and Wang PC.

I develop software for 8086/8088 based machines and I want to do windows, too. I'm enclosing $4.50 for postage and handling. Please send me your free demo disk. My business card is attached. (Offer expires December 31, 1984)

Computer:_____
Name_____
Company_____
Address_____
City_____ State_____ Zip_____

VSI

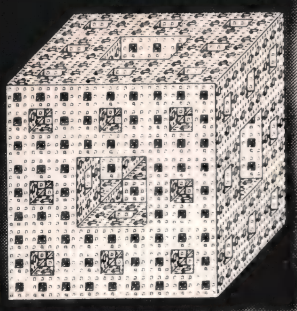
AMBER SYSTEMS, INC.
1171 S. Sunnyvale-Saratoga Road
San Jose CA 95129
(408) 996-1883

Circle **no. 1** on reader service card.

Streams Listing (Text begins on page 50)

```
SCR #219
  0 ( L. L. Odette          6-6-84                          MVP-FORTH )
  1 \ Stream Constructor and Selector Words
  2
  3 : MAKE-HEAD
  4    HERE 10 + [COMPILE] LITERAL     \ Compile IF clause address
  5    COMPILE SWAP
  6    COMPILE BRANCH HERE 0 , 2       \ Delay is branch around IF
  7    [COMPILE] IF ; IMMEDIATE        \ Rest looks like IF
  8
  9 : MAKE-TAIL   [COMPILE] ELSE ; IMMEDIATE
 10
 11 : END-STREAM  [COMPILE] THEN       \ Terminate If..Else..Then
 12    COMPILE EXIT [COMPILE] THEN ;   \ Terminate delaying branch
 13    IMMEDIATE
 14
 15 :  HEAD DROP >R 1 ;           : TAIL  DROP >R 0 ;


SCR #220
  0 ( L. L. Odette          6-6-84                          MVP-FORTH )
  1
  2 : MYSELF   LATEST PFA CFA , ; IMMEDIATE
  3
  4 \  Misc. Stream manipulation words. Each Stream is in the form:
  5 \              (n-2 elements),pointer,n...
  6 \  where the count (n) is on the top of the parameter stack.
  7 \  *** Translation to machine code is recommended ***
  8
  9 : STREAM-DUP   DUP 1+ DUP 1  DO DUP PICK SWAP LOOP  DROP ;
 10
 11 : STREAM-OVER  DUP DUP 2+ PICK   SWAP OVER + 1+ SWAP
 12                0 DO DUP PICK SWAP LOOP DROP ;
 13
 14 : STREAM-SWAP  DUP DUP 2+ PICK   SWAP OVER + 1+ SWAP
 15                0 DO DUP ROLL SWAP LOOP DROP ;


SCR #221
  0 ( L. L. Odette          6-6-84                          MVP-FORTH )
  1
  2 \ More misc. Stream manipulation words
  3
  4 : STREAM-DROP  DUP   0 DO DROP LOOP ;
  5
  6 : STREAM->R  R> OVER DUP
  7              BEGIN   4 ROLL >R   1- DUP 0= UNTIL
  8              DROP >R >R ;
  9
 10 : STREAM-R>  R> R>
 11              BEGIN   R> SWAP   1- DUP 0= UNTIL DROP
 12              DUP 1+ ROLL >R ;
 13
 14
 15
OK
```

(Continued on page 60)



WALTZ LISP (TM)

*The one and only **adult** Lisp system for CP/M users.*

Waltz Lisp is a very powerful and complete implementation of the Lisp programming language. It includes features previously available only in large Lisp systems. In fact, Waltz is substantially compatible with *Franz* (the Lisp running under *Unix*), and is similar to *MacLisp*. Waltz is perfect for Artificial Intelligence programming. It is also most suitable for general applications.

Much faster than other microcomputer Lisps. • Long integers (up to 611 digits). Selectable radix • True dynamic character strings. Full string operations including fast matching/extraction. • Flexibly implemented random file access. • Binary files. • Standard CP/M devices. • Access to disk directories. • Functions of type lambda (expr), nlambda (fexpr), lexpr, macro. • Splicing and non-splicing character macros. • User control over all aspects of the interpreter. • Built-in prettyprinting and formatting facilities. • Complete set of error handling and debugging functions including user programmable processing of undefined function references. • Virtual function definitions. • Optional automatic loading of initialization file. • Powerful CP/M command line parsing. • Fast sorting/merging using user defined comparison predicates. • Full suite of mapping functions, iterators, etc. • Assembly language interface. • Over 250 functions in total. • The best documentation ever produced for a micro Lisp (300+ full size pages, hundreds of illustrative examples).

Waltz Lisp requires CP/M 2.2, Z80 and 48K RAM (more recommended). All common 5'' and 8'' disk formats available.

PROCODE (TM) INTERNATIONAL

15930 SW Colony Pl.
Portland, OR 97224

Unix* Bell Laboratories.
CP/M* Digital Research Corp.

Version 4.4 (Now includes Tiny Prolog written in Waltz Lisp.)

$169*

*Manual only: $30 (refundable with order). All foreign orders: add $5 for surface mail, $20 for airmail. COD add $3. Apple CP/M and hard sector formats add $15.

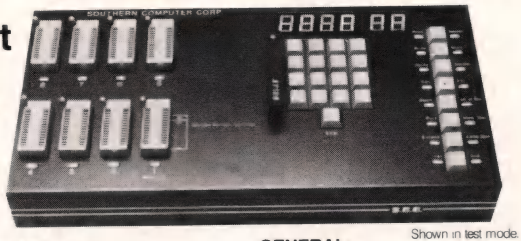
Call free **1-800-LIP-4000** Dept. #11
In Oregon and outside USA call 1-503-684-3000

Circle **no. 60** on reader service card.

The Cost Efficient EPROM Programmer

$995.00 COMPLETE

Dealer inquiries welcome.



Shown in test mode.

DISPLAY:
- Bright 1″ high display system
- Progress indicated during programming
- Error messages

KEYBOARD:
- Full travel entry keys
- Auto repeat
- Illuminated function indicators

INTERFACE:
- RS-232C for data transfer
- 110-19.2K baud
- X-on X-off control of serial data

FUNCTIONS:
- Fast and standard programming algorithms
- Single key commands
- Search finds data strings up to 256 bytes long
- Electronic signatures for easy data error I.D.
- "FF" skipping for max programming speed
- User sets memory boundaries
- 15 commands including move, edit, fill, search, etc. functions
- Extended mode reads EPROM sets

GENERAL:
- Stand alone operation, *external terminal not needed for full command set*
- Total support
- 28 pin sockets
- Faulty EPROMS indicated at socket
- Programs 1 to 128K devices
- Built in diagnostics
- No calibration required
- No personality modules to buy
- Programs new CMOS EPROMS
- Printer interface option
- Complete with 128K buffer

ALSO AVAILABLE FROM SCC:

The Cost Efficient Erasing Units

FIVE TIMES THE CAPACITY OF OTHER UNITS, FOR LESS THAN $200!

FEATURES INCLUDE:

- Unique wave design
- Efficient bulb design
- All-steel, heavy duty design
- Quick erasure time
- Efficient
- Reliable
- Safe
- Affordable and economical
- Portable, easy to use
- EPROMS
- Micro computer
- Industrial design
- Production environment ready
- Timer included

Three Models Available:

EU-156...**over 150 chips**
$195.00

EU-312...**over 300 chips**
$359.95

EU-1050...**over 1000 chips (EPROM or Micro Computer)**
CALL!

QUICK DELIVERY ON ALL PRODUCTS!

FOR FURTHER INFORMATION ON SCC's COST EFFICIENT PROGRAMMERS AND ERASING UNITS CALL

SOUTHERN COMPUTER CORPORATION

3720 N. Stratford Rd., Atlanta, GA 30342, 404-231-5363

Circle **no. 84** on reader service card.

DeSmet C

The fastest 8088 C Compiler available

FULL DEVELOPMENT PACKAGE
- C Compiler
- Assembler
- Linker and Librarian
- Full-Screen Editor
- Newsletter for bugs/updates

SYMBOLIC DEBUGGER
- Monitor and change variables by name using C expressions
- Multi-Screen support for debugging PC graphics and interactive systems
- Optionally display C source during execution
- Breakpoint by Function and Line #

COMPLETE IMPLEMENTATION
- **Both** 1.0 **and** 2.0 DOS support
- Everything in K&R (incl. STDIO)
- Intel assembler mnemonics
- **Both** 8087 **and** Software Floating Point

OUTSTANDING PERFORMANCE

Sieve Benchmark

COMPILE	4 Sec.	RAM —
	22 Sec.	FDISK
LINK	6 Sec.	RAM —
	34 Sec.	FDISK
RUN	12 Sec.	
SIZE	8192 bytes	

DeSmet C Development Package $159

To Order Specify:

Machine _____

OS ☐ MS-DOS ☐ CP/M-86

Disk ☐ 8″ ☐ 5¼ SS ☐ 5¼ DS

CWARE CORPORATION

P.O. BOX 710097
San Jose, CA 95171-0097
(408) 736-6905

California residents add sales tax. Shipping: U.S. no charge, Canada add $5, elsewhere add $15. Checks must be on a US Bank and in US Dollars.

Circle **no. 19** on reader service card.

Streams Listing (Listing Continued, text begins on page 58)

```
SCR #222
  0 ( L. L. Odette          6-6-84                          MVP-FORTH )
  1
  2 ( Useful stream manipulations )
  3
  4 : SPLIT   STREAM-DUP TAIL STREAM-SWAP HEAD ;
  5
  6 : DELAY-ADDR   HERE 24 + [COMPILE] LITERAL ; IMMEDIATE
  7
  8 \ Represents stream tail, calculation delayed once
  9 : DELAY-TAIL
 10    OVER  DELAY-ADDR -                    \ Don't delay twice
 11     IF   DUP 2+   MAKE-HEAD  TAIL HEAD
 12                   MAKE-TAIL  TAIL TAIL END-STREAM
 13     ELSE   DDROP TAIL MYSELF   THEN ;
 14
 15 : DELAY-SPLIT   STREAM-DUP DELAY-TAIL STREAM-SWAP HEAD ;

SCR #223
  0 ( L. L. Odette          6-6-84                          MVP-FORTH )
  1
  2 : <FILTER>   ( Stream,Predicate-address . . . Stream )
  3    >R BEGIN                              \ Save predicate and loop
  4        STREAM-DUP HEAD R@ EXECUTE        \ Test head
  5       WHILE TAIL REPEAT                  \ Filter tail while true
  6    R>   OVER 3 +                         \ Recover predicate
  7    MAKE-HEAD    DROP HEAD                \ Compute first element
  8    MAKE-TAIL    >R TAIL R> MYSELF        \ Compute Remainder
  9    END-STREAM ;
 10
 11 : FILTER
 12   -2 ALLOT HERE @   [COMPILE] LITERAL    \ Pass predicate address
 13   COMPILE <FILTER> ; IMMEDIATE           \ to generic filter
 14
 15


SCR #224
  0 ( L. L. Odette          6-6-84                          MVP-FORTH )
  1
  2 : <MAP>   ( Stream,Function-address . . . Stream )
  3   OVER 3 +                              \ Format for constructor
  4   MAKE-HEAD  >R HEAD R> EXECUTE         \ Apply function to head
  5   MAKE-TAIL  >R TAIL R> MYSELF          \ Map tail
  6   END-STREAM ;
  7
  8 : MAP
  9   -2 ALLOT HERE @   [COMPILE] LITERAL    \ Pass function address
 10   COMPILE <MAP> ; IMMEDIATE              \ to generic map
 11
 12
 13
 14
 15


SCR #225
  0 ( L. L. Odette          6-6-84                          MVP-FORTH )
  1
  2 \ Length refers to # of items in (init-value,operator,length)
  3
  4 : ACCUMULATE  ( Init-value,operator,length,stream . . . Stream )
```

```
 5   DUP 1+ PICK OVER + 2+  \ Compute data length for constructor
 6   MAKE-HEAD              \ (init,op,length) looks like a stream
 7    STREAM-DROP DDROP     \ Compute head
 8   MAKE-TAIL
 9    STREAM-DUP DELAY-TAIL STREAM->R \ Save delayed tail
10    DUP 2+ DUP ROLL SWAP ROLL SWAP  \ Get operator,length
11    >R SP@ 3 PICK 2* +  >R >R       \ Save them with stack data
12    HEAD R@ EXECUTE                 \ Accumulate output and input
13    R> SP@ R> SWAP - 2/ R> +  \ Package (init-value,op,length)
14    STREAM-R> MYSELF          \ Remake accumulation
15   END-STREAM ;

SCR #226
  0 ( L. L. Odette         6-6-84                          MVP-FORTH )
  1
  2 : DIVISIBLE-BY   ( stream,divisor . . . filtered-stream )
  3     >R    BEGIN STREAM-DUP HEAD R@ MOD  \ Divide head by divisor
  4     0=    WHILE  TAIL  REPEAT           \ Repeat while divisible
  5     R> OVER 3 +                         \ Set count for stream
  6    MAKE-HEAD    DROP HEAD               \ head is first not div
  7    MAKE-TAIL    >R TAIL R> MYSELF       \ Filter again for tail
  8    END-STREAM ;
  9
 10 : SIEVE        DUP 2+                   \ Set up length
 11    MAKE-HEAD   HEAD                     \ First element is prime
 12    MAKE-TAIL   SPLIT DIVISIBLE-BY MYSELF \ Filter tail by head
 13    END-STREAM ;
 14
 15

SCR #227
  0 ( L. L. Odette         6-6-84                          MVP-FORTH )
  1
  2 \ Assorted words to manipulate the parameter field pointer
  3 \ and parameter stack representation during EXEC
  4
  5 : GET-POINT  ( . . . x,y,parameter-field-pointer )
  6   R> R> R> R>  4 ROLL >R ;
  7
  8 : PUT-POINT  ( x,y,parameter-field-pointer . . . )
  9   R> SWAP >R SWAP >R SWAP >R >R ;
 10
 11 : RE-STREAM  ( Restore count word to stack representation )
 12   R> R> R> R> SWAP >R SWAP >R SWAP >R ;
 13
 14 : UN-STREAM  ( Stack representation count word to return stack )
 15   R> R> R> 4 ROLL >R >R >R >R ;

SCR #228
  0 ( L. L. Odette         6-6-84                          MVP-FORTH )
  1 \ More assorted pointer and return-stack manipulations
  2
  3 : *LIT  ( . . . @-pointer ) \ Perform like LIT
  4     R> GET-POINT 2+ DUP
  5     >R SWAP >R SWAP >R SWAP >R @ ;
  6
  7 : SET-POINT   ( . . . )    \ Set according to literal field
  8     R> R> *LIT
  9     GET-POINT 2- 4 ROLL +   PUT-POINT   >R >R ;
 10
 11 : +2-POINT                 \ Increment pointer
 12     R> R> GET-POINT 2+ PUT-POINT >R >R ;
 13
 14 : -2-POINT                 \ Decrement pointer
 15     R> R> GET-POINT 2- PUT-POINT >R >R ;
```

(Continued on next page)

Streams Listing (Listing Continued, text begins on page 58)

```
SCR #229
  0 ( L. L. Odette          6-6-84                          MVP-FORTH )
  1
  2 \ Replacement words - acting on environments
  3
  4         : *R*   SWAP >R 1- STREAM-SWAP 1+ R> SWAP STREAM-SWAP ;
  5 ( >R ) : *>R  RE-STREAM *R* UN-STREAM ;
  6 ( R> ) : *R>  RE-STREAM STREAM-SWAP *R* STREAM-SWAP UN-STREAM ;
  7 ( R@ ) : *R@  RE-STREAM STREAM-SWAP OVER SWAP 1+ *R*
  8               STREAM-SWAP UN-STREAM ;
  9
 10 ( EXIT ) : *XT   -2-POINT ;
 11 ( LIT )  : *LT   R> *LIT SWAP >R ;
 12
 13 ( 0BRANCH ) : *0BR  0=  IF SET-POINT ELSE +2-POINT THEN ;
 14 ( BRANCH )  : *BR   SET-POINT ;
 15


SCR #230
  0 ( L. L. Odette          6-6-84                          MVP-FORTH )
  1
  2 \ More replacement words
  3
  4 ( <DO> )     : *DO   SWAP RE-STREAM *R* *R* UN-STREAM ;
  5
  6 ( <LOOP> )     : *LOOP  RE-STREAM  STREAM-SWAP UN-STREAM
  7                         1+ OVER OVER - 0=
  8                         IF   DDROP RE-STREAM 2-
  9                              STREAM-SWAP 2+ UN-STREAM +2-POINT
 10                         ELSE RE-STREAM STREAM-SWAP UN-STREAM
 11                              SET-POINT
 12                         THEN ;
 13
 14
 15


SCR #231
  0 ( L. L. Odette          6-6-84                          MVP-FORTH )
  1 \ Using C.E. Eaker's Case construct. Forth Dimensions II(3):37
  2
  3 : RET-STACK  \ Map ret-stack, pntr-control words to equivalent
  4   CASE
  5   [ FIND R>       ] LITERAL OF  [ FIND *R>    ]  LITERAL  ENDOF
  6   [ FIND >R       ] LITERAL OF  [ FIND *>R    ]  LITERAL  ENDOF
  7   [ FIND R@       ] LITERAL OF  [ FIND *R@    ]  LITERAL  ENDOF
  8   [ FIND EXIT     ] LITERAL OF  [ FIND *XT    ]  LITERAL  ENDOF
  9   [ FIND LIT      ] LITERAL OF  [ FIND *LT    ]  LITERAL  ENDOF
 10   [ FIND 0BRANCH  ] LITERAL OF  [ FIND *0BR   ]  LITERAL  ENDOF
 11   [ FIND BRANCH   ] LITERAL OF  [ FIND *BR    ]  LITERAL  ENDOF
 12   [ FIND <DO>     ] LITERAL OF  [ FIND *DO    ]  LITERAL  ENDOF
 13   [ FIND <LOOP>   ] LITERAL OF  [ FIND *LOOP  ]  LITERAL  ENDOF
 14   DUP ENDCASE  ;
 15

SCR #232
  0 ( L. L. Odette          6-6-84                          MVP-FORTH )
  1
  2 : EXEC  \ Accumulate environments with code address stream
```

(Continued on page 66)

dBASE III™


dBASE III™
SYSTEM DISK
Serial No: 100001
Format: IBM PC, XT and 100% compatibles
PC-DOS 2.0
Version: 1.0
ASHTON-TATE
10150 West Jefferson Blvd.
Culver City CA 90230 (213) 204-5570
TELEX 664228 ASHTATE CULV
©Copyright 1984 Ashton-Tate
as an unpublished licensed
proprietary work.
All rights reserved.

More power to you.

Remember the magic you expected when you first purchased a PC?

It's here.

dBASE III™ is the most powerful database management system ever created for 16-bit microcomputers. It pulls every ounce of energy out of your PC and puts it to work.

On top of that, it's fast and it's easy.

You've never seen anything like it.

dBASE III can handle over a billion records per file, limited only by your computer system. You can have up to ten files open, for sophisticated applications programs.

When you have two related files, information in one can be accessed based upon data in the other.

dBASE III now handles procedures, parameter passing and automatic variables. You can include up to 32 procedures in a single file. With lightning speed. Because once a file is opened, it stays open. And procedures are accessed directly.

Easier than ever.

dBASE III uses powerful yet simple commands that are the next best thing to speaking English.

If you're unsure of a command, HELP will tell you what to ask for.

If you don't know what command comes next, a command assistant does. All you have to know is what you want it to do.

Our new tutorial/manual will have you entering and viewing data in minutes rather than reading for hours.

And to make matters easier, you get a full screen report setup for simple information access.

Faster than no time at all.

dBASE III isn't just fast. It's ultra-fast. Operating. And sorting. Even faster, is no sorting. Because dBASE III keeps your records in order, so you really don't have to sort anything. Unless you want to. Then watch out!

What about dBASE II®?

It's still the world's best database management system for 8-bit computers. And it's still the industry standard for accounting, educational, scientific, financial, business and personal applications.

Tap into our power.

For the name of your nearest authorized dBASE III dealer, contact Ashton-Tate, 10150 West Jefferson Boulevard, Culver City, CA 90230. (800) 437-4329, ext. 333. In Colorado, (303) 799-4900.

ASHTON·TATE™

Streams Listing (Listing Continued, text begins on page 58)

```
  3     SWAP >R SWAP >R SP@ >R     \ Save pointer and stack depth info
  4     EXECUTE
  5     R> SP@ - 2/ R> + R> 2+ ;       \ Re-form environment
  6
  7
  8 : PNTR   \ Extract pointer from an environment
  9     >R STREAM-DROP STREAM-DROP R> ;
 10
 11 ( Maps )
 12 : PNTR-MAP    PNTR       MAP ;
 13 : @-MAP       @          MAP ;
 14 : RET-MAP     RET-STACK  MAP ;
 15

SCR #233
  0 ( L. L. Odette          6-6-84                              MVP-FORTH )
  1
  2 \ ENV-EXEC takes initial environment and leaves accumulation
  3
  4 : ENV-EXEC ( Ret-stack-rep,Param-stack-rep,Pfa . . . Stream )
  5       OVER DUP 3 + PICK + 3 +          \ Compute length
  6       [ FIND EXEC ] LITERAL SWAP       \ Compose for accumulate
  7       STREAM-DUP  DUP 2+               \ Dummy stream
  8        MAKE-HEAD  DDROP  END-STREAM    \ Tail never accessed
  9       PNTR-MAP @-MAP RET-MAP           \ Pass through maps
 10       ACCUMULATE ;
 11
 12
 13
 14
 15


SCR #234
  0 ( L. L. Odette          6-6-84                              MVP-FORTH )
  1
  2 \ STEP expects an accumulation based on EXEC and leaves a stream
  3 \ HEAD leaves the head of the input accumulation
  4 \ TAIL leaves a new accumulation
  5
  6 : STEP   DUP 2+                         \ Set stream count
  7      MAKE-HEAD   HEAD
  8      MAKE-TAIL   TAIL HEAD              \ Get first of rest
  9        ENV-EXEC  MYSELF                 \ Compose accumulate & step
 10      END-STREAM ;
 11
 12
 13
 14
 15


SCR #235
  0 ( L. L. Odette          6-6-84                              MVP-FORTH )
  1
  2 ( [environment],number . . .  )
  3 : ENV-DISPLAY   ." WORD  " . CR HEX 5 SPACES
  4   ." Pfa " U. 2 SPACES
  5   ." Parameter Stack -> "  1- ?DUP IF 0 DO U. LOOP THEN 5 SPACES
  6   ." Return Stack -> "    1- ?DUP IF 0 DO U. LOOP THEN
  7   CR DECIMAL ;
```

```
 8
 9
10 : TEST0  ( Test word 0 )
11   ?DUP  IF  0  DO  2*  LOOP  THEN  ;
12
13 : TEST1  ( Test word 1 )
14   BEGIN  1- DUP 0= UNTIL ;
15

SCR #236
 0 ( L. L. Odette          6-6-84                          MVP-FORTH )
 1
 2 : STEPS  ( N steps for 2 words in parallel )
 3   0 DO
 4   SPLIT  1 ENV-DISPLAY  STREAM-SWAP      \ Execute and display
 5   SPLIT  2 ENV-DISPLAY  STREAM-SWAP      \ Execute and display
 6   LOOP ;
 7
 8 \ Set up stream to execute TEST0
 9 1 ( empty return stack )  2 2 3 ( data stack is: 2 2 )
10 ' TEST0   ENV-EXEC  STEP
11
12 \ Set up stream to execute TEST1
13 1 ( empty return stack )  2 2 ( data stack is: 2 )
14 ' TEST1   ENV-EXEC  STEP
15
```

End Listing

A Forth Native-Code Cross Compiler for the MC68000

by Raymond Buvel

If you have an 8-bit microcomputer system and want to experiment with the Motorola MC68000, this cross compiler may interest you. One of the problems an experimenter faces in using a new microprocessor is how to develop programs for it. While one solution to this problem is to purchase a new computer system and a new set of software for each of the processors that interest you, this is obviously expensive. Another solution is to use an existing computer and its software to develop programs for the new processor. Compilers that use an existing computer to produce code for another machine are called cross compilers.

The cross compiler presented here is a native code compiler for the MC68000. It loads on top of a host Forth development system (which doesn't have to be running on a 68000) and compiles a subset of the Forth language. The compiler itself is written in Forth and I have isolated the necessary system-dependent parts to a few clearly identified words. I expect that the code presented here can be easily ported to most Forth environments.

If you've been wondering how to get into programming on the Motorola 68000 chip, this Forth package could be the tool that gets you started.

The compiler produces two types of compiled definitions: macros and subroutines. The macro definitions are essentially extensions to the compiler itself and do not directly produce an executable program. Subroutine definitions generate the executable code, which can consist of macro, variable, constant, and subroutine references. The executable code is stand-alone and can be placed in ROM. The destination for the executable code is left entirely up to the user and can be changed by altering the definition of a single word in the compiler. The compiler does not contain any built-in words for doing I/O; this is left up to the user.

In this article I am assuming the reader is familiar with the Forth language. Those not familiar with Forth should get a copy of the excellent book *Starting FORTH* by Leo Brodie. I used *Starting FORTH* as a reference when I developed this compiler, so the Forth words implemented here work as described there. I have presented the detailed description of the operators supported by the compiler in assembly language. Although you need a familiarity with the MC68000 assembly language to understand the design of the compiler, it is not necessary to know assembly language to use the compiler.

Memory Layout

Before considering the compiler, you should know how it uses the memory of the MC68000 address space. The compiler uses four separate areas of MC68000 address space. The pointers to these areas all are maintained in MC68000 registers, so the user can allocate any area of memory for the various functions by properly setting up the registers. Thus, you need not recompile the program to change the memory map. These areas are described below.

Raymond Buvel, Box 3071, Moscow, ID 83843.

Code Pool

Subroutine definitions place their output code in the code pool and update the code pool pointer variable in the compiler (M68PCODE in Listing Two, screen 9, on page 90). During execution of the resulting code, the MC68000 program counter is the pointer to this area of memory. Only relative addressing is used, so you can relocate the code pool simply by moving the code and starting the program at the proper place.

Variable Pool

The memory used by variables and arrays is allocated relative to the variable pool pointer (A5 in the MC68000). The compiler word M68ALLOT is used to allocate space and maintain even address alignment. To avoid address faults, the value placed in A5 must be even. With that restriction, you may place the variable pool anywhere in memory by setting the value of A5. With an appropriate supervisor program, you can produce reentrant modules with this compiler by using A5 to assign a separate space for the local variables each time the module is called.

Data Stack

MC68000 register A6 points to the memory used by the data stack. The stack is maintained using the auto decrement addressing mode to store information on the stack and the auto increment addressing mode to remove information from the stack. For further information on the workings of the data stack, see the stack operator section of Listing One (page 76).

Return Stack

The hardware stack is used for the return stack because most of the return stack operations then become automatic. A7 is the pointer to the return stack. Since there is both a supervisor and a user hardware stack pointer, you can use modules generated by this compiler for both interrupt service routines and user programs.

Compiler Description

The Forth subset that I chose to implement is for a particular hardware configuration, but you could expand it if you require different I/O. I assume that you have a computer with a Forth system running on it. In the following discussion, I will refer to this computer as the host. I also assume that you are using the MC68000 as a coprocessor. This assumption greatly reduces the complexity of the subset to be implemented. All functions that interact with the terminal can be left out, and the host can handle all the interaction with the operating system and peripherals.

A simple bidirectional communication channel is all that is required between the host and the MC68000. I have chosen to implement the arithmetic, stack, memory access, and control operators found in most Forth systems. You can write I/O routines for data transfer between the MC68000 and the host using the primitives provided, since the MC68000 uses memory-mapped I/O.

The compiler generates machine code, so there is no need for an inner interpreter. Because the MC68000 provides the capability of writing position-independent code, all of the code produced by this compiler is position independent unless the user explicitly forces it to be otherwise. Because the code is kept separate from the variable and stack space, the output from the compiler can be put into ROM. It is sometimes desirable to use programs generated with this compiler in host environments other than Forth. Therefore, I have provided a simple output scheme that allows you to send the output code to any device supported by the host.

To avoid conflict with the Forth definitions in the host, most of the compiler is in a separate vocabulary (named M68K) that is accessed only through the defining words. All definitions created with the compiler also are placed in this vocabulary to prevent accidental reference to them while using the host development system. I have given some of the words normally used in Forth different names to avoid conflicts with the host definitions; I will discuss these differences later.

The compiler produces two basic types of definitions: macros and subroutines. Macro definitions do not generate any output code, but when referenced they store the code that implements the macro in the definition currently being compiled. Subroutine definitions generate output code when defined and generate a subroutine call when referenced in another subroutine definition. A

macro definition may be referenced in another macro or in a subroutine definition, but a subroutine may not be referenced in a macro definition. The macro definition is the basic building block in the compiler, so I will discuss it in detail before considering constant, variable, and array definitions.

Macros

You create a macro in the same way as you do a Forth colon definition, except that you use :M68MAC and ;M68MAC in place of the : and ; of a Forth definition. The body of the definition consists of executable MC68000 machine code or references to macros, variables, constants, and arrays. For examples of macro definitions, see screens 33 – 43 in Listing Two and the examples below.

Defining a macro activates the M68K vocabulary, creates a Forth header in the dictionary of the host, and reserves space for the code length. The host Forth is in execution mode, so any words referenced in the body of a macro definition are executed immediately. Terminating the macro definition stores the length of the code segment and reactivates the Forth vocabulary. Any subsequent reference to the macro copies the code contained within the macro body into the host dictionary at the location HERE, then the dictionary pointer is updated to point to the memory location following the code.

To illustrate this process, Figure 1 (at left) shows the definition of the macro 2*. First :M68MAC is used to start the definition, create the Forth header for 2* (I am being deliberately vague about the form of the header because that depends on the particular implementation of Forth being used), and allocate space for the code length. Next the macro DUP is called; it copies the code for performing a DUP function into the host dictionary at the location HERE and updates the dictionary pointer by two. Then the macro + is called; it copies its code into the dictionary and updates the dictionary pointer by four. Finally ;M68MAC is used to terminate the definition and compute and store the macro length in the two bytes following the header. In this case, the length is six bytes.

```
:M68MAC 2* DUP + ;M68MAC
```

2*	Dictionary header for the macro 2*
00 06	Length of macro code segment
3D 16	MC68000 machine code for DUP
30 1E	MC68000 machine code for +
D1 56	

Figure 1
A macro definition and the resulting dictionary entry

```
5 M68CON #5          (Defines the constant #5)
```

#5	Dictionary header for the macro #5
00 04	Length of macro code segment
3D 3C	MOVE.W #5, –(A6)
00 05	Pushes the value 5 onto data stack

Figure 2
A constant definition and the resulting dictionary entry

```
5 M68CARY EXAMP      (Defines the byte array EXAMP)
```

EXAMP	Dictionary header for the macro EXAMP
00 06	Length of macro code segment
30 3C	Code to add the variable pool relative
00 20	address of the array (20 hex) to the
D1 56	index value on the data stack.

Variable pool pointer:
Before = 0020; After = 0026

Figure 3
An array definition, the resulting dictionary entry, and the effect on the variable pool pointer (Note all values are in hex)

Constants and Variables

Single- and double-precision constants are compiled as macros containing a single MC68000 instruction to push the value of the constant onto the data stack. For example, the word M68CON is used to define a single-precision constant in Figure 2 (at left). The value of the constant is taken from the host stack and stored in the macro as part of a move-immediate instruction (see Listing One). The word M68DCON is the same, except that it involves a double-precision value.

Variables are defined as single-precision constants that push the variable pool relative address onto the stack for use with the fetch and store operations. Note that the variable pool relative address is a 16-bit signed integer, so the variable pool can be no longer than 32K bytes. The word M68VAR defines a single-precision variable, while the word M68DVAR is for double precision.

Arrays are defined as macros that

take the index off the stack, compute the variable pool relative address of that element, then leave the result on the stack. The compiler supports arrays whose elements are either byte, single precision, or double precision. The words M68CARY, M68ARY, and M68DARY, respectively, define these data types. Figure 3 (page 70) shows the definition of a byte array containing five elements. The variable pool pointer is shown before and after the definition of the array. Note that the compiler maintains alignment on word boundaries to avoid address exceptions when the code is executed.

Subroutines

Defining a subroutine activates the M68K vocabulary and creates a Forth header in the dictionary of the host. The code pool relative address of the subroutine is then stored as the first entry of the definition. Compilation proceeds in the same way as in a macro definition, except that references to subroutine definitions are also allowed. When the definition is terminated, a return from subroutine instruction is compiled, the code pool pointer M68PCODE is updated, and the code is sent to the output file and deleted from the dictionary. This leaves only the header and the code pool relative address of the subroutine in the dictionary.

Subsequent reference to the subroutine uses the code pool relative address to compute the relative address required in a branch to the subroutine instruction. Note that the branch instructions on the MC68000 restrict your program to 32K bytes because all of the subroutine calls are back branches; forward referencing is not supported in this compiler.

Since the code pool relative address of a subroutine is stored at the start of the definition, you may reference a subroutine recursively. You must exercise care when doing this, however. Subroutine calls do not create local variables, so a subroutine that stores a value in a variable may not operate properly when called recursively. I recommend keeping all variables on the stack in recursive subroutines. When you do this, make sure the data stack is large enough; because there is no check for stack overflow, something will be clobbered if the data stack space is too small.

Figure 4 (page 72) illustrates the process of subroutine compilation. The word :M68K is used to start the definition and create the Forth header for 4*; this also sets the code pool relative address of 4* (in this case, it is zero). Next the macro 2* is called twice (this macro 2* is the one defined in Figure 1, not the one actually implemented in the compiler, which is more efficient). Each time 2* is called, the code implementing it is sorted in the host dictionary, and the dictionary pointer is updated. Then ;M68K is used to terminate the definition by compiling a subroutine return instruction, adding the length of the subroutine to the code pool pointer, copying the code to the output file, and deleting the code from the dictionary.

Installation

To install the compiler, you need either a FIG Forth or a Forth-79 system. The installation on a Forth-79 system is a little more involved, so I will cover the FIG installation first. Listing Two contains the complete source for the compiler; you must somehow get these 35 screens into your Forth system. (See the section on availability at the end of this article.)

You must customize a few words in the compiler to your system. In screen 12, a word called HIGH-BYTE takes the top entry off the stack and returns the high byte. You must replace this definition with the code to accomplish this task; it may be necessary to use a code definition on your system. The word M68OUT in screen 18 can be designed to send the generated code to whatever output device or file you want (refer to the note in that screen). The word M68OUT currently prints the code on the screen. If you want to use the external reference capability, you will need to modify the definition in screen 20 to send the output where you want it. If you do not want that feature, simply delete screen 20 entirely.

The compiler is loaded in three sections. The basic compiler and error checking routines are loaded from screens 8 – 24. The program control and looping operations with their associated error checking are contained in screens 25 – 33. The macros that im-

plement the operators supported by the compiler are in screens 34 – 43.

For a Forth-79 system, the above applies, but you must do some additional work. I have used the FIG Forth word ENDIF instead of the Forth-79 word THEN, so on a Forth-79 system the definition of ENDIF given in Listing Two, screen 44, should be used. I have also used the FIG word ⟨BUILDS in a ⟨BUILDS . . . DOES⟩ construction; see screen 44 for a discussion of a definition for ⟨BUILDS. With these two definitions, the compiler should run on a Forth-79 system. However, I cannot guarantee that this is the case. Because my system is a combination FIG Forth with Forth-79 extension, my testing may not have picked up all of the problems. If you have difficulty with this, either get in touch with me or send in a letter to the editor.

Using the Compiler

Before explaining how to use the compiler, I want to point out some of the difficulties you will encounter. Although this compiler uses a subset of Forth, you will find that you must modify most Forth programs before the compiler will accept them. Obviously, you must remove and/or replace constructions that are not supported by the compiler, but many programming techniques used in Forth also will not work because the output code is nothing like the indirect threaded code used in most Forth implementations. Therefore, such practices as modifying the values of constants on the fly will not work, nor can you compile things into the dictionary at runtime since there is no dictionary.

The compiler runs as a collection of words in the host Forth system, but I have not rewritten the Forth word NUMBER to be sensitive to the state of the M68K compiler. Therefore, a number contained within a definition will not be compiled into the definition automatically. Instead it goes on the host stack and must be compiled into the definition with the word LITERAL (or DLITERAL in the case of double precision). Also note that you must use at least one subroutine definition to get the code to the output file. This should all become clear as I go through the example in Listing Three (page 104).

Listing Three, screen 8, shows the Forth implementation of a benchmark program (Jim Gilbreath, *Byte*, September 1981, page 190); I will use this as a reference to show how the Forth code must be modified to compile properly. Screen 9 shows the same program for the M68K compiler. The first thing to notice is the difference in the definitions of the constants. Zero is used several times in the program. To avoid using LITERAL so many times, I defined a constant #0 in the M68K vocabulary and used that wherever a zero was used. (I could have chosen the name 0 for this constant, since it is redefined only when the M68K vocabulary is active, but that would have made the example more confusing.)

The constant SIZE is used in two ways in the program. SIZE is used as a Forth constant when allocating array space and as an M68K constant within the definition of DO-PRIME. It is defined twice, once in the Forth vocabulary using CONSTANT and once in the M68K vocabulary using M68CON. The definition of the variable FLAGS is a little different from screen 8: the M68K compiler follows the convention in *Starting FORTH* and does not provide initialized variables. Note that the constants 1 and 3 used inside DO-PRIME are compiled using LITERAL, as all numbers inside a definition must be if they are not defined as constants. The resulting code for either a constant or literal is identical; only the compilation process is different. I used :M68MAC to define DO-PRIME because I wanted the entire program to be a single subroutine.

Screen 10 contains an initialization routine required to set the pointers used by the code; it also contains a subroutine TEST, which causes the output code to be generated. TEST also causes the benchmark to be iterated 10 times to conform to the requirements of Gilbreath's test. Note that, because the prime count cannot be printed as in the Forth version, it is simply dropped from the stack. You might think that you could eliminate the prime count from the program itself, but that

```
:M68K 4* 2* 2* ;M68K

   4*      Dictionary header for the subroutine 4*
  00 00    Relative address of subroutine

Code pool pointer:
Before = 0000; After = 000E

Code sent to the output file:
        2*          |        2*          |  RTS instruction
--------------------|--------------------|------------------
3D 16 30 1E D1 56   | 3D 16 30 1E D1 56  |  4E 75
```

Figure 4
A subroutine definition, the resulting dictionary entry, and the output code

Source	Code size	Time (10 iter.)	Comments
Screen 8	109 bytes	85 sec	Z80 Forth @ 3.5Mhz
Screen 9	220 bytes	8.7 sec	68000 @ 10Mhz with 2 wait states
Screen 11	216 bytes	8.1 sec	68000 @ 10Mhz with 2 wait states
Assembler	74 bytes	2.1 sec	68000 @ 10Mhz with 2 wait states

Figure 5
Results of benchmark tests shown in Listing Three

B. G. MICRO

P. O. Box 280298 Dallas, Texas 75228
(214) 271-5546

Big Computer Mfg. Makes $900,000 Goof!!

COMPUTER/DISK DRIVE SWITCHING POWER SUPPLY

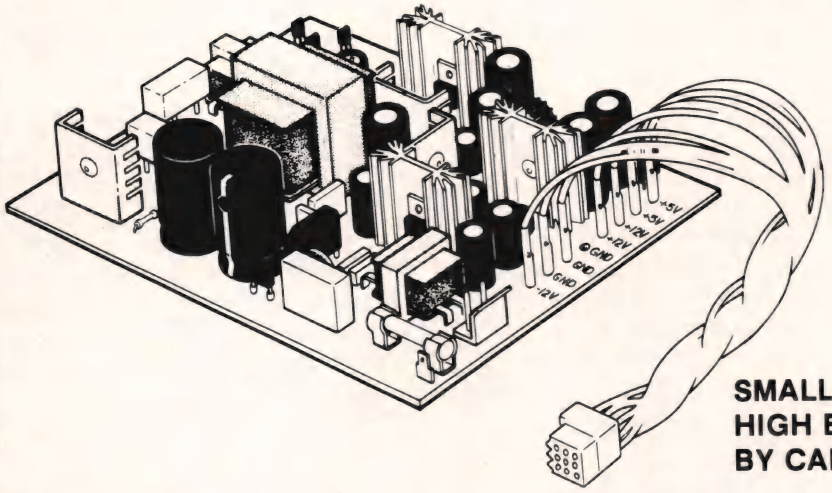
ORIGINAL OEM COST $72 EACH!

ORIGINALLY DESIGNED TO RUN A Z-80 BASED SINGLE BOARD COMPUTER WITH TWO 5-1/4 IN. DISK DRIVES AND CRT MONITOR.

BRAND NEW: UNUSED! **$37.50 EA**

3 FOR $95.00

ADD $1.50 PER UNIT FOR UPS



SPECS:	+ 5VDC 5 AMPS MAX
	#1 + 12 VDC 2.8 AMPS MAX
	#2 + 12 VDC 2.0 AMPS MAX
	-12 VDC .5 AMPS MAX
INPUT: 115 or 230 VAC 60Hz	

SMALL SIZE: 6-1/8 x 7-3/8 In.
HIGH EFFICIENCY SWITCHER MFG. BY CAL. DC IN *USA*!

The poor Purchasing Agent bought about 10 times as many of these DC switchers as his company would ever use! We were told that even in 10,000 piece lots they paid over $72 each for these multi-output switchers. When this large computer manufacturer discontinued their Z-80 Computer, guess what the Big Boss found in the back warehouse; several truckloads of unused $72.00 power supplies. Fortunately we heard about the deal and made the surplus buy of the decade. Even though we bought a huge quantity, please order early to avoid disappointment. Please do not confuse these high quality American made power supplies with the cheap import units sold by others.

TERMS: Orders over $50 add 85¢ insurance. No COD. Tex. Res. Add 6% Sales Tax. Subject to prior sale. Foreign orders: US funds only. We cannot ship to Mexico. Foreign countries other than Canada add $6 per board shipping.

would give a false representation of the execution time of the benchmark.

Screen 11 is the same program as screen 9, rewritten to use the array features of the compiler and to remove a couple of the inefficiencies built into the original program. Screen 12, which is identical to screen 10, is here simply to compile screen 11 without using an indirect LOAD. Also shown in Listing Three is an assembly language version of the benchmark program used for timing and code size comparisons. Figure 5 (page 72) shows the results of the benchmark. Note that we pay a fairly heavy penalty for using a stack-oriented langauge. The programming convenience is worth it in most cases, however.

Figure 6 (page 74) is a description of the words that perform the compilation operations. The list is organized functionally rather than alphabetically. The word described is listed to the left, followed by the host stack image; to the right is an example of the proper usage of the word. Listing One contains the assembly language source for the Forth words supported by this compiler. That listing also, includes a short description of the supported words. The reader should refer to Listing One and *Starting FORTH* to clear up any confusion concerning these definitions. Note that Listing One also describes several operators that are not standard Forth. Operators for doing absolute memory references are described in the memory and I/O section. Absolute subroutine calls and jumps are described in the control operations section.

Debugging Your Programs

No easy way exists to debug programs written for this compiler, so I recommend the following development procedure:

(1) *Write and debug the program in Forth.* Your Forth development system provides a good environment for debugging programs that you develop for this compiler. It is very important at this stage to avoid using any operations not supported by the compiler, since you will just have to remove them later. You should avoid using embedded literals in your definitions—they are rather messy to take care of later.

(2) *Translate the program into a form acceptable to the compiler.* If you have avoided using embedded literals in your definitions, the only changes should be replacing the : and ; as appropriate. You can take care of the emedded literals by defining each one as a constant or using the word LITERAL (or DLITERAL).

(3) *Compile the program and load it into your MC68000 computer.* Make sure that registers A5, A6, and A7 are set properly (either by a supervisor program, by hand, or by using the load instructions provided in the compiler).

The program should now operate properly. If not, step (2) is the most likely place to find the errors. I have frequently encountered errors in resolving program control structures, errors that occur because an embedded literal has not been compiled. This type of error is picked up in the compile phase, so incorrect code is not produced.

Final Comments

This compiler could form the basis for a modular programming environment for the MC68000 microprocessor. The modifications necessary would not be very complicated. To make the compiler generate modular code, you would have to add a new word to the basic compiler. This word should reset all of the compiler variables in screen 9 to their original style and generate an appropriate header to permit the operating system to load and execute the module. Another word to terminate a module and check for errors would be required; the error checking code in the existing compiler can be used as a guide. Note that the absolute addressing operators must be used for all global variables. I recommend that global variables be avoided and that all parameters passed from one module to another be passed on the data stack.

I would like to see floating-point arithmetic added to the compiler, but I

:M68K (--) **:M68K xxxx**

Creates a header for the subroutine word xxxx in the M58K vocabulary and sets the variables M68ENTRY and M68PFA. Reference to xxxx within a subroutine definition generates a branch to subroutine using the PC relative addressing mode. The word xxxx may only be referenced within a subroutine definition. Any other usage will produce an error message. So long as no side effects occur, the word xxxx may be referenced recursively. Note that the code may be put into ROM because the stack and variable space are kept separate from the code.

;M68K (--)

Terminates the construction of a subroutine definition and sends the code to the output file. The code for the subroutine is deleted from the host dictionary after it is written out, and only the code pool relative address of the subroutine is retained.

:M68MAC (--) **:M68MAC xxxx**

Creates a header for the macro word xxxx in the M68K vocaulary and sets the compiler variable M68PFA. Reference to xxxx within a definition copies the compiled code into the host dictionary. A macro word may be referenced within the definition of another macro word or within the definition of a subroutine word.

;M68MAC (--)

Terminates the construction of a macro type word, encloses the code, and updates the compiler variables.

M68CON (n --) **n M68CON xxxx**

Defines a macro word xxxx that pushes the value n onto the stack when xxxx is executed. The value n must be on the host stack when M68CON is referenced.

M68DCON (d --) **d M68DCON xxxx**

Defines a double-precision constant in the same way as M68CON.

Figure 6

am unlikely to do it anytime soon. I would also like to have a MC68000 assembler built into the compiler, so machine code does not have to be typed in hex. (The Forth assembler developed by Michael Perry—*Dr. Dobb's Journal*, September 1983—might be adapted for this purpose.) I encourage anyone who is interested to work on these extensions and publish their work in *Dr. Dobb's* for the rest of us to use.

Please note that I have copyrighted this compiler. However, I am releasing it for personal use and nonprofit distribution. If you sell my compiler as an integral part of a commercial software package, I expect a small royalty. You may sell any code produced by the compiler without notifying me or paying royalties. Other than that, feel free to use it as you see fit and give it to whomever you like. You may also want to send it on to your favorite user group. All I ask is that you acknowledge the source.

Availability

By the time you read this, the compiler should be available from the SIG/M User Group, Box 2085, 177 Hadley Ave., Clifton, NJ 07015 (Disks are 8-inch CP/M and sell for less than $10). The compiler is available on the following RCPM system: Thousand Oaks Technical RCP/M Sysop: Trevor Marshall; System 1: (805) 492-5472; System 2: (805) 493-1495.

I would prefer that you get the compiler from one of the above sources, but I will provide an 8-inch CP/M SSSD disk (sorry, no other formats) for $25. [*We'll keep you updated on other sources as we hear of them. — Ed.*]

DDJ

Reader Ballot
Vote for your favorite feature/article.
Circle Reader Service **No. 195**.

M68ALLOT (n --) **n M68ALLOT**

Allocates n bytes of space in the variable pool by updating the variable pool pointer variable M68PVAR. Note that this word maintains even-byte alignment so that address exceptions will not occur on 16- and 32-bit memory references. Also note that it is an error if the variable pool becomes longer than 32K bytes, but the compiler will not report this as an error—incorrect code would be generated.

M68VAR (--) **M68VAR xxxx**

Defines a single-precision variable by using the pointer M68PVAR as the parameter n for M68CON. The pointer M68PVAR is then updated by two bytes using M68ALLOT. Execution of the word xxxx leaves the variable pool relative address of the variable on the top of the stack.

M68DVAR (--) **M68DVAR xxxx**

Defines a double-precision variable in the same way as M68VAR except that the pointer M68PVAR is updated by four bytes.

M68ARY (n --) **n M68ARY xxxx**

Defines a single-precision array xxxx that is n elements long. The word xxxx is defined as a macro that takes an element number off the top of the stack and leaves the variable pool relative address of that element.

M68CARY (n --) **n M68CARY xxxx**

Defines a byte array in the same manner as M68ARY.

M68DARY (n --) **n M68DARY xxxx**

Defines a double-precision array in the same manner as M68ARY.

EXTERNAL (--) **EXTERNAL xxxx**

Create an external reference xxxx that contains the code pool relative address of the last subroutine word that was defined. The user can customize this word to produce a file containing an external reference list. Currently, this word creates a constant xxxx in the Forth vocabulary of the host.

Figure 6
Description of compiling words

Forth Compiler (Text begins on page 68)

Listing One

```
                ;
                ;  This file contains the assembler language code for all of the
                ;  operations in the M68K compiler.
                ;
                ;  Register usage....
                ;       A7 - Hardware and return stack pointer
                ;       A6 - Data stack pointer
                ;       A5 - Pointer to variable pool
                ;       A4 - Reserved for future use
                ;
                ;       All other registers are free to be used by any word that needs them
                ;       and are to be considered as altered across word boundaries.
                ;
                ;*******************************************************************************
                ;
                ;  Arithmetic operations
                ;
                ;*******************************************************************************
                ;
                ;  +             ( n1 n2 -- sum )
                ;
301E                 MOVE.W   (A6)+,D0                ;Get n2
D156                 ADD.W    D0,(A6)                 ;n1 + n2
                ;
                ;  -             ( n1 n2 -- dif )        n1-n2
                ;
301E                 MOVE.W   (A6)+,D0                ;Get n2
9156                 SUB.W    D0,(A6)                 ;n1 - n2
                ;
                ;  *             ( n1 n2 -- prod )
                ;
301E                 MOVE.W   (A6)+,D0                ;Get n2
C1D6                 MULS     (A6),D0                 ;n2 * n1
3C80                 MOVE.W   D0,(A6)
                ;
                ;  /             ( n1 n2 -- quot )       n1/n2
                ;
4C9E 0003            MOVEM.W  (A6)+,D0/D1             ;Get operands sign extended
83C0                 DIVS     D0,D1                   ;n1/n2
3D01                 MOVE.W   D1,-(A6)
                ;
                ;  */            ( n1 n2 n3 -- n-result )  n1*n2/n3
                ;
321E                 MOVE.W   (A6)+,D1                ;Get n3
301E                 MOVE.W   (A6)+,D0                ;Get n2
C1D6                 MULS     (A6),D0                 ;n2*n1 -> D0
81C1                 DIVS     D1,D0                   ;n2*n1/n3 -> D0
3C80                 MOVE.W   D0,(A6)
                ;
                ;  /MOD          ( u1 u2 -- u-rem u-quot )
                ;
4280                 CLR.L    D0
321E                 MOVE.W   (A6)+,D1                ;Get u2
301E                 MOVE.W   (A6)+,D0                ;Get u1
80C1                 DIVU     D1,D0                   ;u1/u2
4840                 SWAP     D0                      ;Interchange remainder and quotient
2D00                 MOVE.L   D0,-(A6)                ;Return both on stack



                ;
                ;  MOD           ( u1 u2 -- u-rem )
                ;
4280                 CLR.L    D0
321E                 MOVE.W   (A6)+,D1                ;Get u2
301E                 MOVE.W   (A6)+,D0                ;Get u1
80C1                 DIVU     D1,D0                   ;u1/u2
4840                 SWAP     D0                      ;Interchange remainder and quotient
3D00                 MOVE.W   D0,-(A6)                ;Return remainder on stack
```

```
                ;
                ;  */MOD        ( u1 u2 u3 -- u-rem u-result )  u1*u2/u3
                ;
321E                 MOVE.W  (A6)+,D1                ;Get u3
301E                 MOVE.W  (A6)+,D0                ;Get u2
C0DE                 MULU    (A6)+,D0                ;u2*u1 -> D0
80C1                 DIVU    D1,D0                   ;u2*u1/u3 -> D0
4840                 SWAP    D0
2D00                 MOVE.L  D0,-(A6)
                ;
                ;  U*           ( u1 u2 -- ud )
                ;
301E                 MOVE.W  (A6)+,D0
C0DE                 MULU    (A6)+,D0
2D00                 MOVE.L  D0,-(A6)
                ;
                ;  U/MOD        ( ud u1 -- u-rem u-quot )
                ;
321E                 MOVE.W  (A6)+,D1                ;Get u1
201E                 MOVE.L  (A6)+,D0                ;Get ud
80C1                 DIVU    D1,D0                   ;ud/u1
4840                 SWAP    D0
2D00                 MOVE.L  D0,-(A6)
                ;
                ;  1+           ( n -- n+1 )
                ;
5256                 ADDQ.W  #1,(A6)
                ;
                ;  1-           ( n -- n-1 )
                ;
5356                 SUBQ.W  #1,(A6)
                ;
                ;  2+           ( n -- n+2 )
                ;
5456                 ADDQ.W  #2,(A6)
                ;
                ;  2-           ( n -- n-2 )
                ;
5556                 SUBQ.W  #2,(A6)
                ;
                ;  2*           ( n -- n*2 )
                ;
E1D6                 ASL     (A6)
                ;
                ;  2/           ( n -- n/2 )
                ;
E0D6                 ASR     (A6)
                ;
                ;  ABS          ( n -- abs(n) )
                ;
4A56                 TST.W   (A6)                    ;Test for negative
6C02                 BGE.S   ABS                     ;Skip next instruction if not
4456                 NEG.W   (A6)
               ABS
                ;
                ;  DABS         ( d -- abs(d) )
                ;
4A96                 TST.L   (A6)                    ;Test for negative
6C02                 BGE.S   DABS                    ;Skip next instruction if not
4496                 NEG.L   (A6)
               DABS
                ;
                ;  NEGATE       ( n -- -n )
                ;
4456                 NEG.W   (A6)
                ;
                ;  DNEGATE      ( d -- -d )
                ;
4496                 NEG.L   (A6)
                ;
                ;  D+           ( d1 d2 -- d-sum )
                ;
201E                 MOVE.L  (A6)+,D0
D196                 ADD.L   D0,(A6)
```

(Continued on next page)

Forth Compiler (Listing Continued, text begins on page 68)
Listing One

```
                ;
                ;  D-              ( d1 d2 -- d-diff )      d1-d2
                ;
201E                 MOVE.L   (A6)+,D0
9196                 SUB.L    D0,(A6)
                ;
                ;
                ;*****************************************************************************
                ;
                ;  Stack manipulation
                ;
                ;*****************************************************************************
                ;
                ;  DROP            ( n -- )
                ;
548E                 ADDQ.L   #2,A6
                ;
                ;  2DROP           ( d -- )
                ;
588E                 ADDQ.L   #4,A6
                ;
                ;  SWAP
                ;
2016                 MOVE.L   (A6),D0
4840                 SWAP     D0
2C80                 MOVE.L   D0,(A6)
                ;
                ;  2SWAP           ( d1 d2 -- d2 d1 )
                ;
2016                 MOVE.L   (A6),D0
2CAE 0004            MOVE.L   4(A6),(A6)

2D40 0004            MOVE.L   D0,4(A6)
                ;
                ;  DUP             ( n -- n n )
                ;
3D16                 MOVE.W   (A6),-(A6)
                ;
                ;  2DUP            ( d -- d d )
                ;
2D16                 MOVE.L   (A6),-(A6)
                ;
                ;  OVER            ( n1 n2 -- n1 n2 n1 )
                ;
3D2E 0002            MOVE.W   2(A6),-(A6)
                ;
                ;  2OVER           ( d1 d2 -- d1 d2 d1 )
                ;
2D2E 0004            MOVE.L   4(A6),-(A6)
                ;
                ;  >R              ( n -- )          Store on return stack
                ;
3F1E                 MOVE.W   (A6)+,-(A7)
                ;
                ;  R>              ( -- n )          Remove from return stack
                ;
3D1F                 MOVE.W   (A7)+,-(A6)
                ;
                ;  I               ( -- n )          Copies top of return stack
                ;
3D17                 MOVE.W   (A7),-(A6)
                ;
                ;  I'              ( -- n )          Copies second item on return stack
                ;
```

(Continued on page 80)

Forth Compiler (Listing Continued, text begins on page 68)

Listing One

```
3D2F 0002          MOVE.W  2(A7),-(A6)
                   ;
                   ;  J             ( -- n )        Copies third item on return stack
                   ;
3D2F 0004          MOVE.W  4(A7),-(A6)
                   ;
                   ;  Push a constant onto the stack       ( -- n )
                   ;
3D3C 0000          MOVE.W  #0,-(A6)
                   ;
                   ;  Push a double constant onto the stack        ( -- d )
                   ;
2D3C 0000          MOVE.L  #0,-(A6)
     0000
                   ;
                   ;
                   ;*****************************************************************************
                   ;
                   ;  Memory and I/O operations
                   ;
                   ;*****************************************************************************
                   ;
                   ;  Note.. all references to memory are relative to A5 unless otherwise
                   ;         specified.
                   ;


                   ;
                   ;  !               ( n adr -- )
                   ;                  Store in variable
                   ;
301E               MOVE.W  (A6)+,D0
3B9E 0000          MOVE.W  (A6)+,0(A5,D0.W)
                   ;
                   ;  @               ( adr -- n )
                   ;                  Get from variable
                   ;
3016               MOVE.W  (A6),D0
3CB5 0000          MOVE.W  0(A5,D0.W),(A6)
                   ;
                   ;  C!              ( c adr -- )
                   ;                  Store in variable
                   ;
301E               MOVE.W  (A6)+,D0
321E               MOVE.W  (A6)+,D1
1B81 0000          MOVE.B  D1,0(A5,D0.W)
                   ;
                   ;  C@              ( adr -- c )
                   ;                  Get from variable
                   ;
3016               MOVE.W  (A6),D0
4241               CLR.W   D1
1235 0000          MOVE.B  0(A5,D0.W),D1
3C81               MOVE.W  D1,(A6)
                   ;
                   ;  2!              ( d adr -- )
                   ;                  Store in variable
                   ;
301E               MOVE.W  (A6)+,D0
2B9E 0000          MOVE.L  (A6)+,0(A5,D0.W)
                   ;
                   ;  2@              ( adr -- d )
                   ;                  Get from variable
                   ;
301E               MOVE.W  (A6)+,D0
2D35 0000          MOVE.L  0(A5,D0.W),-(A6)
                   ;
                   ;  +!              ( n adr -- )
                   ;                  Add n to the loaction pointed to by adr
```

SOFTWARE TOOLS FOR ADVANCED PROGRAMMERS

Dr. Dobb's Journal — Reader Service Card

To obtain information about products or services mentioned in this issue, circle the appropriate number listed below. Use bottom row to vote for best article in issue. This card is valid for 90 days from issue date. Use only one card per person.

Sept. 1984, No. 95

Name _____ Phone (___) _____

Address _____

City/State/Zip _____

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54
55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81
82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100	101	102	103	104	105	106	107	108
109	110	111	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127	128	129	130	131	132	133	134	135
Articles:	190	191	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207	208	209	210	211	212	213	214	

Comments:

Purchase of Magazine:

1 ☐ Subscription
2 ☐ Computer Store
3 ☐ Newsstand
4 ☐ Bookstore
5 ☐ Passed on by friend/colleague
6 ☐ Other _____

SOFTWARE TOOLS FOR ADVANCED PROGRAMMERS

Dr. Dobb's Journal — Reader Service Card

To obtain information about products or services mentioned in this issue, circle the appropriate number listed below. Use bottom row to vote for best article in issue. This card is valid for 90 days from issue date. Use only one card per person.

Sept. 1984, No. 95

Name _____ Phone (___) _____

Address _____

City/State/Zip _____

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54
55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81
82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100	101	102	103	104	105	106	107	108
109	110	111	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127	128	129	130	131	132	133	134	135
Articles:	190	191	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207	208	209	210	211	212	213	214	

Comments:

Purchase of Magazine:

1 ☐ Subscription
2 ☐ Computer Store
3 ☐ Newsstand
4 ☐ Bookstore
5 ☐ Passed on by friend/colleague
6 ☐ Other _____

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY CARD
FIRST CLASS PERMIT NO. 756 MENLO PARK, CA

POSTAGE WILL BE PAID BY ADDRESSEE

SOFTWARE TOOLS FOR ADVANCED PROGRAMMERS
Dr. Dobb's Journal

P. O. Box 27809
San Diego, CA 92128

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY CARD
FIRST CLASS PERMIT NO. 756 MENLO PARK, CA

POSTAGE WILL BE PAID BY ADDRESSEE

SOFTWARE TOOLS FOR ADVANCED PROGRAMMERS
Dr. Dobb's Journal

2464 EMBARCADERO WAY
PALO ALTO, CA 94303

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY CARD
FIRST CLASS PERMIT NO. 756 MENLO PARK, CA

POSTAGE WILL BE PAID BY ADDRESSEE

SOFTWARE TOOLS FOR ADVANCED PROGRAMMERS
Dr. Dobb's Journal

2464 EMBARCADERO WAY
PALO ALTO, CA 94303

```
                ;
301E                MOVE.W  (A6)+,D0
321E                MOVE.W  (A6)+,D1
D375 0000           ADD.W   D1,0(A5,D0.W)
                ;
                ;   M68ARY xxxx  ( n -- )        defines an array xxxx n words long
                ;
                ;
                ;   xxxx         ( n -- adr )    returns the address of the n-th element of xxxx
                ;
303C 0000           MOVE.W  #0,D0           ;Array base address
D056                ADD.W   (A6),D0         ;n + address
D156                ADD.W   D0,(A6)         ;2*n + address
                ;
                ;   M68CARY xxxx ( n -- )        defines an array xxxx n bytes long


                ;
                ;
                ;   xxxx         ( n -- adr )    returns the address of the n-th element of xxxx
                ;
303C 0000           MOVE.W  #0,D0           ;Array base address
D156                ADD.W   D0,(A6)         ;n + address
                ;
                ;   M68DARY xxxx ( n -- )        defines an array xxxx n double words long
                ;
                ;
                ;   xxxx         ( n -- adr )    returns the address of the n-th element of xxxx
                ;

303C 0000           MOVE.W  #0,D0           ;Array base address
3216                MOVE.W  (A6),D1         ;n
E541                ASL.W   #2,D1           ;4*n
D041                ADD.W   D1,D0           ;4*n + address
3C80                MOVE.W  D0,(A6)
                ;
                ;   FILL         ( adr n b -- )
                ;                Fills n bytes of memory beginning at the variable pool
                ;                relative address with the value b.
                ;
301E                MOVE.W  (A6)+,D0        ;Get b
321E                MOVE.W  (A6)+,D1        ;Get n
305E                MOVEA.W (A6)+,A0        ;Get variable pool relative address
D1CD                ADDA.L  A5,A0           ;Compute actual address
6002                BRA.S   $02             ;Enter loop at proper point
10C0        $01     MOVE.B  D0,(A0)+        ;Store b and increment address
51C9 FFFC   $02     DBF     D1,$01          ;Repeat n times
                ;*****************************************************************************
                ;
                ; Note.. the following words reference absoute memory addresses.  They should
                ;        only be used to reference data and I/O devices that are fixed and
                ;        outside the environment of the compiler.  Under NO conditions should
                ;        these operations be used to reference data structures created by the
                ;        compiler.  The compiler data structures are relocatable and there is
                ;        no easy way to find the current location of these data structures.
                ;        The other operators provided above are much more convenient and
                ;        preserve the relocatability.
                ;
                ;
                ;   AW!          ( n short -- )
                ;                Store n at the location specified by the short address.
                ;
305E                MOVEA.W (A6)+,A0        ;Get address
309E                MOVE.W  (A6)+,(A0)      ;Store n
                ;
                ;   AW@          ( short -- n )
                ;                Get n from the location specified by the short address.
                ;
3056                MOVEA.W (A6),A0         ;Get address
3C90                MOVE.W  (A0),(A6)       ;Get n
                ;
                ;   AL!          ( n long -- )
                ;                Store n at the location specified by the long address.
                ;
                ;
```

(Continued on next page)

Forth Compiler (Listing Continued, text begins on page 68)

Listing One

```
205E            MOVEA.L (A6)+,A0        ;Get address
309E            MOVE.W  (A6)+,(A0)      ;Store n
                ;
                ; AL@          ( long -- n )
                ;               Get n from the location specified by the long address.
                ;
205E            MOVEA.L (A6)+,A0        ;Get address
3D10            MOVE.W  (A0),-(A6)      ;Get n
                ;
                ; CAW!         ( c short -- )
                ;               Store c at the location specified by the short address.
                ;
305E            MOVEA.W (A6)+,A0        ;Get address
301E            MOVE.W  (A6)+,D0        ;Get c
1080            MOVE.B  D0,(A0)         ;Store c
                ;
                ; CAW@          ( short -- c )
                ;               Get c from the location specified by the short address.
                ;
3056            MOVEA.W (A6),A0         ;Get address
4240            CLR.W   D0
1010            MOVE.B  (A0),D0         ;Get c
3C80            MOVE.W  D0,(A6)
                ;
                ; CAL!          ( c long -- )
                ;               Store c at the location specified by the long address.
                ;
205E            MOVEA.L (A6)+,A0        ;Get address
301E            MOVE.W  (A6)+,D0        ;Get c
1080            MOVE.B  D0,(A0)         ;Store c
                ;
                ; CAL@          ( long -- c )
                ;               Get c from the location specified by the long address.
                ;
205E            MOVEA.L (A6)+,A0        ;Get address
4240            CLR.W   D0
1010            MOVE.B  (A0),D0         ;Get c
3D00            MOVE.W  D0,-(A6)
                ;
                ; 2AW!          ( d short -- )
                ;               Store d at the location specified by the short address.
                ;
305E            MOVEA.W (A6)+,A0        ;Get address
209E            MOVE.L  (A6)+,(A0)      ;Store d
                ;
                ; 2AW@          ( short -- d )
                ;               Get d from the location specified by the short address.
                ;
305E            MOVEA.W (A6)+,A0        ;Get address
2D10            MOVE.L  (A0),-(A6)      ;Get d
                ;
                ; 2AL!          ( d long -- )
                ;               Store d at the location specified by the long address.
                ;
205E            MOVEA.L (A6)+,A0        ;Get address
209E            MOVE.L  (A6)+,(A0)      ;Store d
                ;


                ; 2AL@          ( long -- d )
                ;               Get d from the location specified by the long address.
                ;
2056            MOVEA.L (A6),A0         ;Get address
2C90            MOVE.L  (A0),(A6)       ;Get d
                ;
                ; AFILL         ( long_adr n b -- )
                ;               Fills n bytes of memory beginning at the long absolute
                ;               address with the value b.
```

(Continued on page 84

ST-FORTH $35

For your IBM PC, XT, or Compatible Computer

A complete FORTH development system for beginners or experienced users

- 100% FORTH-83 Standard System
- Written entirely in FORTH. All source code is provided
- Powerful easy-to-use editor
- Complete Assembler with Intel Mnemonics
- Extensive MS-DOS/PC-DOS Interface (DOS 1.x or 2.x)
- Choice of Block Storage: mapped to disk, into DOS file, or in RAM
- Save environment to a DOS .COM file
- Beginners pkg. $60 CA residents add sales tax

Sunset Technology
1954 Menalto Ave.
Menlo Park, CA 94025
(415) 325-3680

Circle **no. 86** on reader service card.

Checks & Balances

Setting the standard in checkbook and budget management for home and office.

•Display, add and change entries with full on-screen editing options•
•Single entry system•
•Unlimited categories•
•Divide entries between multiple categories•
•Flexible search and printing•
•Commands in plain English•

For CP/M-80 2.2, MS-DOS and PC-DOS

Only $74.95 ($54.95 without check printing capability)
Visa and MasterCard accepted
Dealer/Distributor/Bundling rates available

CDE SOFTWARE

2463 McCready Avenue • Los Angeles, CA 90039
Telephone: (213) 661-2031

Circle **no. 10** on reader service card.

MicroMotion

MasterFORTH

It's here — the next generation of MicroMotion Forth.

- Meets all provisions, extensions and experimental proposals of the FORTH-83 International Standard.
- Uses the host operating system file structure (APPLE DOS 3.3 & CP/M 2.x).
- Built-in micro-assembler with numeric local labels.
- A full screen editor is provided which includes 16 x 64 format, can push & pop more than one line, user definable controls, upper/lower case keyboard entry, A COPY utility moves screens within & between lines, line stack, redefinable control keys, and search & replace commands.
- Includes all file primitives described in Kernigan and Plauger's Software Tools.
- The input and output streams are fully redirectable.
- The editor, assembler and screen copy utilities are provided as relocatable object modules. They are brought into the dictionary on demand and may be released with a single command.
- Many key nucleus commands are vectored. Error handling, number parsing, keyboard translation and so on can be redefined as needed by user programs. They are automatically returned to their previous definitions when the program is forgotten.
- The string-handling package is the finest and most complete available.
- A listing of the nucleus is provided as part of the documentation.
- The language implementation exactly matches the one described in FORTH TOOLS, by Anderson & Tracy. This 200 page tutorial and reference manual is included with MasterFORTH.
- Floating Point & HIRES options available.
- Available for APPLE II/II+/IIe & CP/M 2.x users.
- MasterFORTH – $100.00. FP & HIRES –$40.00 each
- Publications
  - FORTH TOOLS – $20.00
  - 83 International Standard – $15.00
  - FORTH-83 Source Listing 6502, 8080, 8086 – $20.00 each.

Contact:
MicroMotion
12077 Wilshire Blvd., Ste. 506
Los Angeles, CA 90025
(213) 821-4340

Circle **no. 42** on reader service card.

Forth Compiler (Listing Continued, text begins on page 68)

Listing One

```
                ;
301E                    MOVE.W  (A6)+,D0        ;Get b
321E                    MOVE.W  (A6)+,D1        ;Get n
205E                    MOVEA.L (A6)+,A0        ;Get absolute address
6002                    BRA.S   $02             ;Enter loop at proper point
10C0            $01     MOVE.B  D0,(A0)+        ;Store b and increment address
51C9 FFFC       $02     DBF     D1,$01          ;Repeat n times
                ;
                ;*****************************************************************************
                ;
                ;  Comparison operations
                ;
                ;*****************************************************************************
                ;
                ;  MIN          ( n1 n2 -- n-min )
                ;
301E                    MOVE.W  (A6)+,D0                ;n2
3216                    MOVE.W  (A6),D1                 ;n1
B041                    CMP.W   D1,D0                   ;n2-n1
6F02                    BLE.S   MIN
C141                    EXG     D0,D1                   ;Swap if D1 < D0
3C80            MIN     MOVE.W  D0,(A6)
                ;
                ;  MAX          ( n1 n2 -- n-max )
                ;
301E                    MOVE.W  (A6)+,D0                ;n2
3216                    MOVE.W  (A6),D1                 ;n1
B041                    CMP.W   D1,D0                   ;n2-n1
6C02                    BGE.S   MAX
C141                    EXG     D0,D1                   ;Swap if D1 > D0
3C80            MAX     MOVE.W  D0,(A6)
                ;
                ;  =            ( n1 n2 -- f )          if n1 = n2 then f is true
                ;
301E                    MOVE.W  (A6)+,D0                ;n2
321E                    MOVE.W  (A6)+,D1                ;n1
B240                    CMP.W   D0,D1                   ;n1-n2
57C0                    SEQ     D0
0240 0001               ANDI.W  #1,D0
3D00                    MOVE.W  D0,-(A6)
                ;
                ;  <            ( n1 n2 -- f )          if n1 < n2 then f is true
                ;
301E                    MOVE.W  (A6)+,D0                ;n2
321E                    MOVE.W  (A6)+,D1                ;n1
B240                    CMP.W   D0,D1                   ;n1-n2
5DC0                    SLT     D0

0240 0001               ANDI.W  #1,D0
3D00                    MOVE.W  D0,-(A6)
                ;
                ;  >            ( n1 n2 -- f )          if n1 > n2 then f is true
                ;
301E                    MOVE.W  (A6)+,D0                ;n2
321E                    MOVE.W  (A6)+,D1                ;n1
B240                    CMP.W   D0,D1                   ;n1-n2
5EC0                    SGT     D0
0240 0001               ANDI.W  #1,D0
3D00                    MOVE.W  D0,-(A6)
                ;
                ;  D=           ( d1 d2 -- f )          if d1 = d2 then f is true
                ;
201E                    MOVE.L  (A6)+,D0                ;d2
221E                    MOVE.L  (A6)+,D1                ;d1
B280                    CMP.L   D0,D1                   ;d1-d2
57C0                    SEQ     D0
0240 0001               ANDI.W  #1,D0
3D00                    MOVE.W  D0,-(A6)
```

```
                ;
                ;  D<            ( d1 d2 -- f )          if d1 < d2 then f is true
                ;
201E                 MOVE.L   (A6)+,D0                ;d2
221E                 MOVE.L   (A6)+,D1                ;d1
B280                 CMP.L    D0,D1                   ;d1-d2
5DC0                 SLT      D0
0240 0001            ANDI.W   #1,D0
3D00                 MOVE.W   D0,-(A6)
                ;
                ;  D>            ( d1 d2 -- f )          if d1 > d2 then f is true
                ;
201E                 MOVE.L   (A6)+,D0                ;d2
221E                 MOVE.L   (A6)+,D1                ;d1
B280                 CMP.L    D0,D1                   ;d1-d2
5EC0                 SGT      D0
0240 0001            ANDI.W   #1,D0
3D00                 MOVE.W   D0,-(A6)
                ;
                ;  0=            ( n -- f )              if n = 0 then f is true
                ;  Alternate name is NOT
                ;
4A5E                 TST.W    (A6)+
57C0                 SEQ      D0
0240 0001            ANDI.W   #1,D0
3D00                 MOVE.W   D0,-(A6)
                ;
                ;  0<            ( n -- f )              if n < 0 then f is true
                ;
4A5E                 TST.W    (A6)+
5DC0                 SLT      D0
0240 0001            ANDI.W   #1,D0
3D00                 MOVE.W   D0,-(A6)
                ;
                ;  0>            ( n -- f )              if n > 0 then f is true
                ;
4A5E                 TST.W    (A6)+

5EC0                 SGT      D0
0240 0001            ANDI.W   #1,D0
3D00                 MOVE.W   D0,-(A6)
                ;
                ;  D0=           ( d -- f )              if d = 0 then f is true
                ;
4A9E                 TST.L    (A6)+
57C0                 SEQ      D0
0240 0001            ANDI.W   #1,D0
3D00                 MOVE.W   D0,-(A6)
                ;
                ;  D0<           ( d -- f )              if d < 0 then f is true
                ;
4A9E                 TST.L    (A6)+
5DC0                 SLT      D0
0240 0001            ANDI.W   #1,D0
3D00                 MOVE.W   D0,-(A6)
                ;
                ;  D0>           ( d -- f )              if d > 0 then f is true
                ;
4A9E                 TST.L    (A6)+
5EC0                 SGT      D0
0240 0001            ANDI.W   #1,D0
3D00                 MOVE.W   D0,-(A6)
                ;
                ;  AND           ( u1 u2 -- and )
                ;
301E                 MOVE.W   (A6)+,D0
C156                 AND.W    D0,(A6)
                ;
                ;  OR            ( u1 u2 -- or )
                ;
301E                 MOVE.W   (A6)+,D0
8156                 OR.W     D0,(A6)
```

(Continued on next page)

Forth Compiler (Listing Continued, text begins on page 68)

Listing One

```
                ;
                ;  XOR          ( u1 u2 -- xor )        Exclusive OR
                ;
301E                 MOVE.W  (A6)+,D0
B156                 EOR.W   D0,(A6)
                ;
                ;  1'S          ( u -- compl )          One's compliment
                ;
4656                 NOT.W   (A6)
                ;
                ;*****************************************************************************
                ;
                ;  Control operations
                ;
                ;*****************************************************************************
                ;
                ;  Note.. all control structures use the PC relative addressing mode
                ;         this makes the code position independent.
                ;
                ;
                ;  IF           ( f -- )
                ;               Takes an entry off the stack and branches if the value

                ;               is FALSE (0).
                ;
4A5E                 TST.W   (A6)+                   ;Remove and test flag
6700 ****            BEQ     ELSE
                ;
                ;  ELSE         ( -- )
                ;               Branches around the else part and provides a target for the
                ;               false branch of the if part.
                ;
6000 ****            BRA     ENDIF
                ELSE
                ;
                ;  ENDIF        ( -- )
                ;               Provides a target for the branch around the else part, and if
                ;               the else is missing provides a target for the false branch of
                ;               the if part.
                ;
                ENDIF
                ;
                ;  BEGIN        ( -- )
                ;               Provides a target for a branch back from UNTIL, AGAIN, REPEAT.
                ;
                BEGIN
                ;
                ;  UNTIL        ( f -- )
                ;               Takes an entry off the stack and branches to BEGIN if the
                ;               value is FALSE (0).
                ;
4A5E                 TST.W   (A6)+                   ;Remove and test flag
6700 ****            BEQ     BEGIN
                ;
                ;  AGAIN        ( -- )
                ;               Always branches to BEGIN
                ;
6000 ****            BRA     BEGIN
                ;
                ;  WHILE        ( f -- )
                ;               Takes an entry off the stack and branches to REPEAT if the
                ;               value is FALSE (0).
                ;
4A5E                 TST.W   (A6)+                   ;Remove and test flag
6700 ****            BEQ     REPEAT
```

(Continued on page 88)

Forth Compiler (Listing Continued, text begins on page 68)

Listing One

```
                ;
                ;  REPEAT          ( -- )
                ;                  Always jumps back to begin and provides a target for WHILE.
                ;
6000 ****            BRA      BEGIN
                REPEAT
                ;
                ;  DO              ( limit index -- )
                ;                  Remove the index and limit from the data stack and put on
                ;                  the return stack.  Provides a target for LOOP and +LOOP
                ;
2F1E                 MOVE.L   (A6)+,-(A7)
                DO
                ;
                ;  LOOP            ( -- )

                ;                  Increment the index and test for end of the loop.
                ;
5257                 ADDQ.W   #1,(A7)                    ;Increment index
4C97 0003            MOVEM.W  (A7),D0/D1                 ;Get index (D0) and limit (D1)
B041                 CMP.W    D1,D0
6D00 ****            BLT      DO                         ;Continue if (index - limit) < 0
588F                 ADDQ.L   #4,A7                      ;Drop index and limit
                ;
                ;  +LOOP           ( n -- )
                ;                  Add n to the index then
                ;                     IF n > 0 then continue if (index - limit) < 0
                ;                     ELSE continue if (index - limit) >= 0.
                ;
301E                 MOVE.W   (A6)+,D0                   ;Get increment
D157                 ADD.W    D0,(A7)                    ;Update index
4C97 0006            MOVEM.W  (A7),D1/D2                 ;Get index (D1) and limit (D2)
4A40                 TST.W    D0                         ;Test for negative
6E04                 BGT.S    $01
B441                 CMP.W    D1,D2                      ;Test (limit - index)
6002                 BRA.S    $02
B242            $01  CMP.W    D2,D1                      ;Test (index - limit)
6D00 ****       $02  BLT      DO                         ;Continue if (condition tested) < 0
588F                 ADDQ.L   #4,A7                      ;Drop index and limit
                ;
                ;  LEAVE           ( -- )
                ;                  Terminate loop by seting limit equal to index
                ;
3F57 0002            MOVE.W   (A7),2(A7)
                ;
                ;  Routines for accessing external programs and subroutines.
                ;
                ;
                ;  JSR.W           ( short address -- )
                ;                  Jump to subroutine using short address from tos
                ;
305E                 MOVEA.W  (A6)+,A0
4E90                 JSR      (A0)
                ;
                ;  JSR.L           ( long address -- )
                ;                  Jump to subroutine using long address from tos
                ;
205E                 MOVEA.L  (A6)+,A0
4E90                 JSR      (A0)
                ;
                ;  JMP.W           ( short address -- )
                ;                  Jump to location pointed to by short address on stack
                ;
305E                 MOVEA.W  (A6)+,A0
4ED0                 JMP      (A0)
```


(Continued on page 90)

SUPER FORTH 64

By Elliot B. Schneider

TOTAL CONTROL OVER YOUR COMMODORE-64™
USING ONLY *WORDS*

MAKING PROGRAMMING FAST, FUN AND EASY!

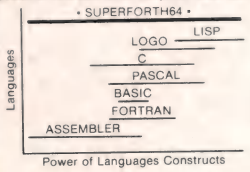
MORE THAN JUST A LANGUAGE...
A complete, fully-integrated program development system.

Home Use, Fast Games, Graphics, Data Acquisition, Business, Music
Real Time Process Control, Communications, Robotics, Scientific, Artificial Intelligence

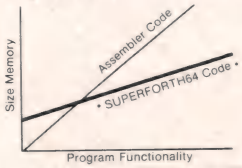
A Powerful Superset of MVPFORTH/FORTH 79 + Ext. for the beginner or professional

- 20 to 600 x faster than Basic
- 1/4 x the programming time
- Easy full control of all sound, hi res. graphics, color, sprite, plotting line & circle
- Controllable SPLIT-SCREEN Display
- Includes interactive interpreter & compiler
- Forth virtual memory
- Full cursor Screen Editor
- Provision for application program distribution without licensing
- FORTH equivalent Kernal Routines
- Conditional Macro Assembler
- Meets all Forth 79 standards+
- Source screens provided
- Compatible with the book "Starting Forth" by Leo Brodie
- Access to all I/O ports RS232, IEEE, including memory & interrupts
- ROMABLE code generator
- MUSIC-EDITOR
- TURTLE GRAPHICS
- SPRITE-EDITOR
- Access all C-64 peripherals including 4040 drive and EPROM Programmer.
- Single disk drive backup utility
- Disk & Cassette based. Disk included
- Full disk usage — 680 Sectors
- Supports all Commodore file types and Forth Virtual disk
- Access to 20K RAM underneath ROM areas
- Vectored kernal words
- TRACE facility
- DECOMPILER facility
- Full String Handling
- ASCII error messages
- FLOATING POINT MATH SIN/COS & SQRT
- Conversational user defined Commands
- Tutorial examples provided, in extensive 261 page cross referenced manual
- INTERRUPT routines provide easy control of hardware timers, alarms and devices
- USER Support

SUPER FORTH 64® *is more powerful than most other computer languages!*



SUPER FORTH 64® *compiled code becomes more compact than even assembly code!*



A SUPERIOR PRODUCT in every way! At a low price of only **$96**

Call:
(415) 651-3160

PARSEC RESEARCH
Drawer 1776, Fremont, CA 94538

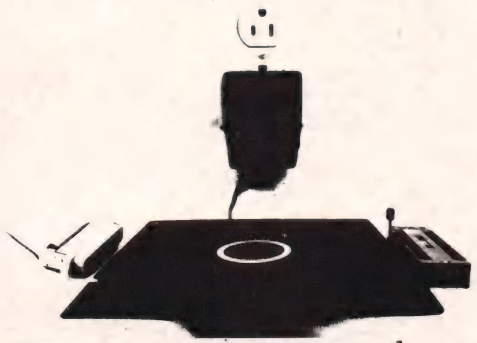
Take this ad to your local dealer.
Phone orders also accepted. Immediate delivery!
Dealer inquiries invited
CA residents must include tax

® PARSEC RESEARCH (Established 1976)

Commodore 64 & VIC-20 TM of Commodore

Circle **no. 54** on reader service card.

THE MOST ADVANCED VERSION OF FORTH IS NOT ON THIS 5 1/4" DISK.



IT'S UNDERNEATH IT!

When you pay $300 for some languages all you get is a $3.00 disk. When you buy the NMIX-0012 RSC-FORTH System from New Micros, Inc. you not only get the language, you get a complete computer system. It has features that you can't expect from a disk based language, like an RS 232 serial interface, 40 individually programmable input/output lines (5 parallel ports), two counter/timers, RAM, ROM and a EEPROM/EPROM programmer. For that matter, disks don't usually come with their own power supplies, and almost none are packaged in rugged metal cases! You would expect a disk to come with hundreds of pages of documentation — our system certainly does. With a disk, you have to tie up your whole computer to run the program. After storing your program in our system, it can be dedicated to a task without the support of your PC, or you can communicate with it over the same serial channel used to program it. On top of all this our FORTH has advanced features, like built-in target compilation, CASE statements, PROM programming words and, believe it or not, disk access functions. The NMIX-0012 is 100 millimeters on a side, smaller than a disk, priced at $290 complete. The "100 Squared"™ is the logical choice! Substantial quantity discounts.

NMI

ORDER NOW FROM: New Micros, Inc
808 Dalworth
Grand Prairie, Texas 75050
(214) 642-5494

Circle **no. 49** on reader service card.

RED

edward k. ream

FULL SCREEN EDITOR
with
FULL SOURCE CODE in C
for
CP/M 68K or CP/M 80

- RED is a powerful yet simple full screen text editor for both programmers and writers.
- RED features 17 commands including block move, block copy, search and substitute.
- RED comes with *full* source code in standard C. RED works as is with the BDS C, Aztec CII and Digital Research Compilers.
- RED supports *all* features of your terminal. You tailor RED to your terminal with an easy-to-use configuration program.
- RED handles files as large as your disk.
- RED is guaranteed. If for any reason you are not satisfied with RED, your money will be refunded promptly.

Price: $95.

Call today for more valuable information:
(608) 231-2952

To order, send a check or money order to:

Edward K. Ream
1850 Summit Avenue
Madison, Wisconsin 53705

Your order will be mailed to you within one week. Sorry, I can not handle credit cards. Please do not send purchase orders unless a check is included. RED is distributed only on 8 inch CP/M format disks, but other disk formats are available through third parties.

Dealer inquiries invited.

Forth Compiler (Listing Continued, text begins on page 68)

Listing One

```
                ;
                ;  JMP.L        ( long address -- )
                ;               Jump to location pointed to by long address on stack
                ;
205E                MOVEA.L (A6)+,A0
4ED0                JMP     (A0)
                ;
                ;

                ;*****************************************************************************
                ;
                ;  Initialization operations
                ;
                ;*****************************************************************************
                ;
                ;  Note.. the following words initialize the registers of the M68000
                ;
                ;
                ;  A5LD         Load the variable pool pointer
                ;
2A7C 0000           MOVEA.L #0,A5
     0000
                ;
                ;  A6LD         Load the data stack pointer
                ;
2C7C 0000           MOVEA.L #0,A6
     0000
                ;
                ;  A7LD         Load the return stack pointer
                ;
2E7C 0000           MOVEA.L #0,A7
     0000
                ;
                ;
                    END
```

End Listing One

Listing Two

```
Screen # 0

 0 ( M68K Cross compiler -- Copyright Notice )
 1 ;S
 2 FORTH based cross compiler for the Motorola 68000 microprocessor
 3
 4 Copyright 1983 by   Raymond L. Buvel
 5                     Box 3071
 6                     Moscow, ID 83843
 7
 8 All rights reserved except as stated below.
 9
10 This compiler may be distributed to anyone provided this
11 copyright notice is included and the distribution is not for
12 profit.  Contact me concerning royalties for commercial
13 distribution.  There is no royalty on code produced with this
14 compiler provided the compiler itself is not SOLD as an integral
15 part of a software package.

Screen # 8

 0 ( M68K Cross Compiler -- Vocabulary definition )
 1 VOCABULARY M68K IMMEDIATE
 2 M68K DEFINITIONS
```

```
 3 HEX
 4 -->
 5 Note.. the compilation words listed below are contained in
 6 the FORTH vocabulary and cause entries to be made in the
 7 M68K vocabulary.
 8
 9 :M68K    :M68MAC M68VAR  M68DVAR M68CON  M68DCON
10 M68ARY  M68DARY M68CARY
11
12
13
14
15

Screen # 9

 0 ( M68K Cross Compiler -- Variable definitions )
 1 M68K DEFINITIONS
 2 ( Code pointer in M68000 -- note relative addressing ! )
 3 0 VARIABLE M68PCODE
 4 ( Variable pool pointer in M68000 -- relative to A5 )
 5 0 VARIABLE M68PVAR
 6 ( Entry point of the subroutine being defined )
 7 0 VARIABLE M68ENTRY
 8 ( Parameter field address [ in HOST ] of word being defined )
 9 0 VARIABLE M68PFA
10 -->
11
12
13
14
15

Screen # 10

 0 ( M68K Cross Compiler -- Variable definitions )
 1 M68K DEFINITIONS
 2 ( Error checking variables )
 3 0 VARIABLE M68?MAC ( True if in a MACRO definition )
 4 0 VARIABLE M68K?   ( True if in a SUBROUTINE definition )
 5 0 VARIABLE M68?PAIRS ( Count of incomplete branching ops. )
 6 -->
 7
 8
 9
10
11
12
13
14
15

Screen # 11

 0 ( M68K Cross Compiler -- Error checking )
 1 M68K DEFINITIONS
 2 : ?M68PAIRS ( Check for unbalanced control structures )
 3         M68?PAIRS @ IF
 4           ." Error! unbalanced control structure "
 5           0 M68?PAIRS ! ABORT ENDIF ;
 6 : ?M68K ( Check for errors in compiling a subroutine )
 7         M68K? @ 0= IF ( Check if compiling a subroutine )
 8           ." Error! not compiling a SUBROUTINE "
 9           ABORT ENDIF ;
10 : ?M68MAC ( Check for errors in compiling a macro )
11         M68?MAC @ 0= IF ( Check if compiling a macro )
12           ." Error! not compiling a MACRO "
13           ABORT ENDIF ;
14 -->
15

Screen # 12

 0 ( M68K Cross Compiler -- Compile constants )
 1 M68K DEFINITIONS
 2 ( n -- c )
```

(Continued on next page)

Forth Compiler (Listing Continued, text begins on page 68)

Listing Two

```
 3 : HIGH-BYTE 8 SHIFT ;  ( Leave high byte of n on stack )
 4 ( n -- )
 5 : $CON  DUP HIGH-BYTE C, C, ; ( Compile const high-byte first )
 6 ( d -- )
 7 : $DCON $CON ( Compile high word )
 8         $CON ; ( Compile low word )
 9 -->
10 Note.. to transport the compiler to other FORTH systems the
11 word HIGH-BYTE must be written so that it takes the number off
12 the top of the stack and leaves the high byte of that number.
13 On some FORTH systems HIGH-BYTE may have to be a CODE
14 definition.
15


Screen # 13

 0 ( M68K Cross Compiler -- Compiling Words )
 1 M68K DEFINITIONS HEX
 2 ( address -- )
 3 : M68MAC ( Compile MACRO code into any definition )
 4         DUP @ SWAP 2+ OVER HERE SWAP CMOVE ALLOT ;
 5 : M68SUB ( Compile SUBROUTINE code into subroutine definition )
 6         ?M68K 61 C, 00 C,  ( BSR  addr )
 7         HERE M68PFA @ 2+ -  ( Compute code length )
 8         M68ENTRY @ + SWAP @ SWAP - ( Compute displacement )
 9         $CON ; ( Compile displacement )
10 -->
11 Note.. the memory image of a MACRO to be compiled is:
12   addr    Number of bytes of code to compile
13   addr+2  Bytes of code to be compiled.
14 The memory image of a SUBROUTINE to be compiled is:
15   addr    Address of subroutine relative to start of code

Screen # 14

 0 ( M68K Cross Compiler -- MACRO Compiling Words  )
 1 FORTH DEFINITIONS
 2 ( Create header and set compiler variables )
 3 : :M68MAC ( Begin a MACRO definition )
 4         [COMPILE] M68K DEFINITIONS
 5         M68K 1 M68?MAC ! <BUILDS HERE M68PFA !
 6         0 , ( Initialize the number of bytes field )
 7         DOES> M68MAC ;
 8 M68K DEFINITIONS
 9 : ;M68MAC ( terminate a MACRO type definition )
10         ?M68PAIRS ?M68MAC 0 M68?MAC ! ( Error check & reset )
11         HERE M68PFA @ 2+ -  ( Compute code length )
12         M68PFA @ !  ( Store in length field )
13         [COMPILE] FORTH DEFINITIONS ;
14 -->
15

Screen # 15

 0 ( M68K Cross Compiler -- Compiling words - constants    )
 1 FORTH DEFINITIONS HEX
 2 : M68CON ( Define a single precision constant )
 3         :M68MAC 3D C, 3C C, ( MOVE.W #const,-[A6] )
 4         M68K $CON ( Compile constant )
 5         ;M68MAC ;
 6 FORTH DEFINITIONS
 7 : M68DCON ( Define a double precision constant )
 8         :M68MAC 2D C, 3C C, ( MOVE.L #const,-[A6] )
 9         M68K $DCON ( Compile double constant )
10         ;M68MAC ;
11 -->
12
```

(Continued on page 94)

Forth Compiler (Listing Continued, text begins on page 68)

Listing Two

```
13
14
15

Screen # 16

 0 ( M68K Cross Compiler -- Compiling words - variables    )
 1 FORTH DEFINITIONS
 2 ( n -- )
 3 : M68ALLOT ( Allot n-bytes in variable pool )
 4         DUP 1 AND IF 1+ ENDIF ( Byte allign )
 5         M68K M68PVAR +! ; ( Update pointer )
 6 FORTH DEFINITIONS
 7 : M68VAR ( Define a single precision variable )
 8         M68K M68PVAR @ 2 M68ALLOT ( Get and update pointer )
 9         M68CON ( Define the address as a constant ) ;
10 FORTH DEFINITIONS
11 : M68DVAR ( Define a double precision variable )
12         M68K M68PVAR @ 4 M68ALLOT ( Get and update pointer )
13         M68CON ( Define the address as a constant ) ;
14 -->
15

Screen # 17

 0 ( M68K Cross Compiler -- SUBROUTINE Compiling Words  )
 1 FORTH DEFINITIONS
 2 ( Create header and set compiler variables )
 3 : :M68K ( Begin a SUBROUTINE definition )
 4         [COMPILE] M68K DEFINITIONS
 5         M68K 1 M68K? ! ( Set to compiling )
 6         <BUILDS HERE M68PFA ! M68PCODE @ DUP
 7         M68ENTRY ! , ( Store subroutine address )
 8         DOES> M68SUB ;
 9 -->
10 Note.. a SUBROUTINE definition may call itself if there are
11 no side effects.  This means that all data altered by the
12 defined word should be on the stack, not stored in variables.
13
14
15

Screen # 18

 0 ( M68K Cross Compiler -- Code output )
 1 M68K DEFINITIONS
 2 ( byte to be sent to code output file -- )
 3 : M68OUT ( Link to the code output file )
 4         BASE @ >R HEX . CR R> BASE ! ;
 5 -->
 6 Note.. the code in the above definition should be replaced
 7 with the appropriate words to send the compiler output to the
 8 code file of your choice.  This could be a disk file, a tape,
 9 your MC68000 computer, a printer, or any other output sink you
10 may want to use.  The protocall is determined by your output
11 word.  The compiler does not assume any protocall so it is a
12 general purpose tool for generating MC68000 code.
13
14
15


Screen # 19

 0 ( M68K Cross Compiler -- SUBROUTINE Compiling Words  )
 1 M68K DEFINITIONS HEX
 2 : ;M68K ( Terminate a SUBROUTINE definition )
 3         ?M68PAIRS ?M68K 0 M68K? ! ( Error check & reset )
 4         4E C, 75 C,  ( Compile an RTS instruction )
```

```
 5         HERE M68PFA @ 2+ -  ( Compute code length )
 6         DUP M68PCODE +! ( Update code pointer )
 7         M68PFA @ 2+ ( Start of compiled code )
 8         SWAP 0 DO
 9                 DUP C@ M68OUT 1+ ( Output a byte of code )
10         LOOP DROP
11         M68PFA @ 2+ DP ! ( Delete code from dictionary )
12         [COMPILE] FORTH DEFINITIONS ;
13 -->
14
15

Screen # 20

 0 ( M68K Cross Compiler -- EXTERNAL )
 1 M68K DEFINITIONS
 2 : $EXTERNAL ( Define entry point as a constant in FORTH voc.
 3         [COMPILE] FORTH DEFINITIONS
 4         M68ENTRY @ CONSTANT ;
 5 FORTH DEFINITIONS
 6 : EXTERNAL ( Compile an external reference )
 7         M68K M68K? @ M68?MAC @ OR
 8         IF ." Can't use EXTERNAL while compiling"
 9            CR ABORT ENDIF
10         $EXTERNAL ;
11 -->
12 Note.. to send the external reference list somewhere else,
13 replace $EXTERNAL with the appropriate word.  Make sure its
14 function is equivalent to the above, i.e. it must take the
15 next word in the input stream as the identifier.

Screen # 21

 0 ( M68K Cross Compiler -- Words - literals )
 1 M68K DEFINITIONS HEX
 2 : LITERAL ( Define a single precision literal )
 3         3D C, 3C C, ( MOVE.W #const,-[A6] )
 4         $CON ; ( Compile constant )
 5 : DLITERAL ( Define a double precision literal )
 6         2D C, 3C C, ( MOVE.L #const,-[A6] )
 7         $DCON ; ( Compile double constant )
 8 : BYTES 0 DO 20 WORD HERE NUMBER DROP C, LOOP ;
 9 -->
10 Note.. Used as n BYTES followed by bytes to be compiled into
11 the HOST dictionary.  This word may be used within a :M68K
12 or :M68MAC definition but NOT within a colon definition.
13
14
15


Screen # 22

 0 ( M68K Cross Compiler -- Compiling words - arrays )
 1 M68K DEFINITIONS HEX
 2 ( adr -- )
 3 : $M68ARY ( Define code for a single precision array )
 4         :M68MAC 30 C, 3C C, ( MOVE.W #const,D0 )
 5         $CON ( Compile address )
 6         D0 C, 56 C, D1 C, 56 C,
 7         ;M68MAC ;
 8 ( adr -- )
 9 : $M68DARY ( Define code for a double precision array )
10         :M68MAC 30 C, 3C C, ( MOVE.W #const,D0 )
11         $CON ( Compile address )
12         32 C, 16 C, E5 C, 41 C, D0 C, 41 C, 3C C, 80 C,
13         ;M68MAC ;
14 -->
15

Screen # 23

 0 ( M68K Cross Compiler -- Compiling words - arrays )
 1 FORTH DEFINITIONS
 2 ( n -- )
 3 : M68ARY ( Define a single precision array n cells long )
```

(Continued on next page)

Forth Compiler (Listing Continued, text begins on page 68)

Listing Two

```
 4         M68K M68PVAR @ ( Get base address )
 5         $M68ARY ( Define the referencing code )
 6         2* M68ALLOT ( Update variable pointer ) ;
 7 FORTH DEFINITIONS
 8 ( n -- )
 9 : M68DARY ( Define a double precision array n cells long )
10         M68K M68PVAR @ ( Get base address )
11         $M68DARY ( Define the referencing code )
12         4 * M68ALLOT ( Update variable pointer ) ;
13 -->
14
15

Screen # 24

 0 ( M68K Cross Compiler -- Compiling words - arrays )
 1 M68K DEFINITIONS HEX
 2 ( adr -- )
 3 : $M68CARY ( Define code for a byte array )
 4         :M68MAC 30 C, 3C C, ( MOVE.W #const,D0 )
 5         $CON ( Compile address )
 6         D1 C, 56 C, ;M68MAC ;
 7 FORTH DEFINITIONS
 8 ( n -- )
 9 : M68CARY ( Define a byte array n cells long )
10         M68K M68PVAR @ ( Get base address )
11         $M68CARY ( Define the referencing code )
12         M68ALLOT ; ( Update variable pointer )
13 ;S
14
15


Screen # 25

 0 ( M68K Cross Compiler -- Control error checking )
 1 M68K DEFINITIONS HEX
 2 ( Error checking codes )
 3 1 CONSTANT $ECD-IF
 4 2 CONSTANT $ECD-BEGIN
 5 3 CONSTANT $ECD-DO
 6 4 CONSTANT $ECD-WHILE
 7 : $ERR-?PAIRS ( Abort if no control structure is started )
 8         M68?PAIRS @ 0=
 9         IF ." No control structure! " ABORT CR ENDIF ;
10 : $ERR-ABT ( Complete error message and abort )
11         ." expected " CR ABORT ;
12 : $ERR-IF ( Abort if no IF structure )
13         $ERR-?PAIRS $ECD-IF -
14         IF ." IF structure " $ERR-ABT ENDIF ;
15 -->

Screen # 26

 0 ( M68K Cross Compiler -- Control error checking )
 1 : $ERR-BEGIN ( Abort if no BEGIN structure )
 2         $ERR-?PAIRS $ECD-BEGIN -
 3         IF ." BEGIN structure " $ERR-ABT ENDIF ;
 4 : $ERR-DO ( Abort if no DO structure )
 5         $ERR-?PAIRS $ECD-DO -
 6         IF ." DO structure " $ERR-ABT ENDIF ;
 7 : $ERR-WHILE ( Abort if no WHILE structure )
 8         $ERR-?PAIRS $ECD-WHILE -
 9         IF ." WHILE structure " $ERR-ABT ENDIF ;
10 -->
11
12
13
```

(Continued on page 98)

NEW! FREE ISSUE of PROFESSIONAL COMPUTING

for Hewlett-Packard Personal Computer Users

NOW – Advance your career by exploiting the full potential of your unique PC!

Imagine how much more productive and more influential you'd be if you could use your Hewlett-Packard PC at its fullest capabilities today. Now there's an authoritative magazine devoted entirely to helping you get ahead by getting the most out of your Hewlett-Packard equipment.

PROFESSIONAL COMPUTING covers it all for you: NEW software . . . NEW applications . . . NEW hardware . . . NEW techniques . . . and much more! All in one, up-to-the-minute, dependable, easy to understand and use source. You'll be aware of every important Hewlett-Packard PC breakthrough as it happens — so you can cash in on it immediately.

You'll look to PROFESSIONAL COMPUTING for timely, useful news, information, and insights on virtually everything related to mastering your Hewlett-Packard PC for business or personal reward. You'll get:

SOFTWARE REVIEWS. New software for various HP models in various formats for word processing, data base management, telecommunications, CAD, and more, is reviewed and evaluated.

INNOVATIVE APPLICATIONS. Practical, how-to information on creative new ways to boost productivity, and solve problems faster.

SOFTWARE LISTINGS. Complete listings of the latest software available for a specific business or technical use — so you can tell at a glance which program is best for you.

HARDWARE. All the newest HP and compatible hardware and peripherals to make your system speedier and more powerful.

TECHNICAL UPDATES. Learn what goes on inside your HP hardware and operating system, and programming tips, so you can trouble-shoot your own problems and expand your output far beyond the routine.

AND MUCH MORE. User exchanges to share solutions, unique new applications . . . answers to readers' questions . . . news and facts on all HP PCs including hand-held, portable and mainstream PCs . . . and more.

Save time and money
Send for your FREE copy today

Whether you're just a beginner or a computer expert, you'll find PROFESSIONAL COMPUTING intriguing, informative, and valuable. It's your single most convenient way to keep on top of Hewlett-Packard PC news you can use.

To get the latest exciting issue of PROFESSIONAL COMPUTING and a no-risk Trial Charter Subscription, simply complete and mail the card or coupon today. Examine PROFESSIONAL COMPUTING at our expense. If you don't think it'll help you be more productive and profitable, simply cancel and owe nothing.

Don't delay! The computer business is moving at break-neck speed — you could be missing a money-making opportunity right now. Do it today!

Exclusive, Experienced

Published solely for Hewlett-Packard PC Users by John Wiley & Sons, one of America's oldest and most distinguished publishers of business and scientific journals and books.

NEW CHARTER OFFER

PROFESSIONAL COMPUTING
John Wiley & Sons, Inc.
P.O. Box 1967
Marion, Ohio 43306

FREE ISSUE

☐ **YES!** I want to advance my career by mastering the full capabilities of my Hewlett-Packard PC. Please send me the latest issue of PROFESSIONAL COMPUTING absolutely free and begin a Trial Charter Subscription at the special rate of $30.00 for one year (6 issues in all). If I'm not delighted, I'll cancel and owe nothing.

☐ Please bill me. ☐ Payment enclosed for faster processing.

Name _____

Address _____

City _____ State _____ Zip _____

EDR9B

Circle **no. 61** on reader service card.

Forth Compiler (Listing Continued, text begins on page 68)
Listing Two

```
14
15

Screen # 27

 0 ( M68K Cross Compiler -- Control structures )
 1 ( adr -- )
 2 : $FOR-RES ( Resolve a foreward branch )
 3         HERE OVER - ( Compute relative address )
 4         SWAP OVER HIGH-BYTE OVER C! ( Store high byte )
 5         1+ C! ; ( Store low byte )
 6 ( adr -- )
 7 : $BAK-RES ( Resolve a back branch )
 8         HERE - ( Compute relative address )
 9         $CON ; ( Compile address )
10 -->
11
12
13
14
15

Screen # 28

 0 ( M68K Cross Compiler -- Control structures )
 1 ( -- adr ecd )
 2 : IF     ( Compile IF structure, leave address to be resolved )
 3          ( and an error checking code )
 4          4A C, 5E C, 67 C, 00 C,
 5          HERE $ECD-IF 1 M68?PAIRS +!
 6          0 , ; ( Leave space for branch address )
 7 : ELSE   ( Compile an ELSE structure )
 8          $ERR-IF 60 C, 00 C,
 9          HERE SWAP ( Save current location and get IF adr )
10          0 , ( Leave space for branch address )
11          $FOR-RES ( Resolve IF branch )
12          $ECD-IF ;
13 : ENDIF ( Resolve an IF structure )
14          $ERR-IF $FOR-RES -1 M68?PAIRS +! ;
15 : THEN   ENDIF ;          -->

Screen # 29

 0 ( M68K Cross Compiler -- Control structures )
 1 ( -- adr ecd )
 2 : BEGIN ( Compile a BEGIN structure )
 3         HERE $ECD-BEGIN 1 M68?PAIRS +! ;
 4 : UNTIL ( Resolve BEGIN .. UNTIL loop )
 5         $ERR-BEGIN 4A C, 5E C, 67 C, 00 C,
 6         $BAK-RES ( Resolve BEGIN branch )
 7         -1 M68?PAIRS +! ;
 8 : AGAIN ( Resolve BEGIN .. AGAIN loop )
 9         $ERR-BEGIN 60 C, 00 C,
10         $BAK-RES ( Resolve BEGIN branch )
11         -1 M68?PAIRS +! ;
12 -->
13
14
15

Screen # 30

 0 ( M68K Cross Compiler -- Control structures )
 1 : WHILE ( Compile WHILE section of loop )
 2         DUP $ERR-BEGIN 4A C, 5E C, 67 C, 00 C,
 3         HERE $ECD-WHILE 0 , ; ( Leave space for address )
 4 : REPEAT ( Resolve BEGIN .. WHILE .. REPEAT loop )
```

```
 5          $ERR-WHILE SWAP $ERR-BEGIN
 6          60 C, 00 C, ( Code for back branch )
 7          SWAP $BAK-RES ( Resolve BEGIN branch )
 8          $FOR-RES ( Resolve WHILE branch )
 9          -1 M68?PAIRS +! ;
10 -->
11
12
13
14
15

Screen # 31

 0 ( M68K Cross Compiler -- Control structures )
 1 : DO ( Compile a DO structure )
 2          2F C, 1E C,
 3          HERE $ECD-DO 1 M68?PAIRS +! ;
 4 : LOOP ( Terminate a DO .. LOOP )
 5          $ERR-DO  52 C, 57 C, 4C C, 97 C, 00 C, 03 C,
 6          B0 C, 41 C, 6D C, 00 C,
 7          $BAK-RES ( Resolve DO branch )
 8          58 C, 8F C,  ( Drop index and limit )
 9          -1 M68?PAIRS +! ;
10 -->
11
12
13
14
15

Screen # 32

 0 ( M68K Cross Compiler -- Control structures )
 1 : +LOOP ( Terminate a DO .. +LOOP )
 2          $ERR-DO  30 C, 1E C, D1 C, 57 C, 4C C, 97 C,
 3          00 C, 06 C, 4A C, 40 C, 6E C, 04 C, B4 C, 41 C,
 4          60 C, 02 C, B2 C, 42 C, 6D C, 00 C,
 5          $BAK-RES ( Resolve DO branch )
 6          58 C, 8F C,  ( Drop index and limit )
 7          -1 M68?PAIRS +! ;
 8
 9 :M68MAC LEAVE 4 BYTES 3F 57 00 02 ;M68MAC
10 -->
11
12
13
14
15

Screen # 33

 0 ( M68K Cross Compiler -- Control structures )
 1 :M68MAC JSR.W 4 BYTES 30 5E 4E 90 ;M68MAC
 2 :M68MAC JSR.L 4 BYTES 20 5E 4E 90 ;M68MAC
 3 :M68MAC JMP.W 4 BYTES 30 5E 4E D0 ;M68MAC
 4 :M68MAC JMP.L 4 BYTES 20 5E 4E D0 ;M68MAC
 5 ;S
 6
 7
 8
 9
10
11
12
13
14
15

Screen # 34

 0 ( M68K Cross Compiler -- Initialization words )
 1 M68K DEFINITIONS HEX
 2 ( d -- )
 3 : A5LD  ( Load variable pool pointer )
 4          2A C, 7C C, $DCON ;
```

(Continued on next page)

Forth Compiler (Listing Continued, text begins on page 68)

Listing Two

```
 5 : A6LD  ( Load data stack pointer )
 6         2C C, 7C C, $DCON ;
 7 : A7LD  ( Load return stack pointer )
 8         2E C, 7C C, $DCON ;
 9 -->
10 Note.. to create true modular programs there should be an
11 operating system that loads the appropriate registers and then
12 calls the module.  In that case these words should be discarded
13 since the address is determined at compile time instead of run
14 time.
15

Screen # 35

 0 ( M68K Cross Compiler -- Arithmetic words )
 1 HEX
 2 :M68MAC + 4 BYTES 30 1E D1 56 ;M68MAC
 3 :M68MAC - 4 BYTES 30 1E 91 56 ;M68MAC
 4 :M68MAC * 6 BYTES 30 1E C1 D6 3C 80 ;M68MAC
 5 :M68MAC / 8 BYTES 4C 9E 00 03 83 C0 3D 01 ;M68MAC
 6 :M68MAC D+ 4 BYTES 20 1E D1 96 ;M68MAC
 7 :M68MAC D- 4 BYTES 20 1E 91 96 ;M68MAC
 8 :M68MAC */ A BYTES 32 1E 30 1E C1 D6 81 C1 3C 80 ;M68MAC
 9 :M68MAC /MOD 8 BYTES 42 80 32 1E 30 1E 80 C1
10              4 BYTES 48 40 2D 00 ;M68MAC
11 :M68MAC MOD 8 BYTES 42 80 32 1E 30 1E 80 C1
12             4 BYTES 48 40 3D 00 ;M68MAC
13 :M68MAC */MOD 8 BYTES 32 1E 30 1E C0 DE 80 C1
14               4 BYTES 48 40 2D 00 ;M68MAC
15 -->

Screen # 36

 0 ( M68K Cross Compiler -- Arithmetic words )
 1 :M68MAC U* 6 BYTES 30 1E C0 DE 2D 00 ;M68MAC
 2 :M68MAC U/MOD A BYTES 32 1E 20 1E 80 C1 48 40 2D 00 ;M68MAC
 3 :M68MAC 1+ 2 BYTES 52 56 ;M68MAC
 4 :M68MAC 1- 2 BYTES 53 56 ;M68MAC
 5 :M68MAC 2+ 2 BYTES 54 56 ;M68MAC
 6 :M68MAC 2- 2 BYTES 55 56 ;M68MAC
 7 :M68MAC 2* 2 BYTES E1 D6 ;M68MAC
 8 :M68MAC 2/ 2 BYTES E0 D6 ;M68MAC
 9 :M68MAC NEGATE 2 BYTES 44 56 ;M68MAC
10 :M68MAC MINUS NEGATE ;M68MAC
11 :M68MAC DNEGATE 2 BYTES 44 96 ;M68MAC
12 :M68MAC DMINUS DNEGATE ;M68MAC
13 :M68MAC ABS 6 BYTES 4A 56 6C 02 44 56 ;M68MAC
14 :M68MAC DABS 6 BYTES 4A 96 6C 02 44 96 ;M68MAC
15 -->


Screen # 37

 0 ( M68K Cross Compiler -- Stack manipulation )
 1 :M68MAC DROP 2 BYTES 54 8E ;M68MAC
 2 :M68MAC 2DROP 2 BYTES 58 8E ;M68MAC
 3 :M68MAC DUP 2 BYTES 3D 16 ;M68MAC
 4 :M68MAC 2DUP 2 BYTES 2D 16 ;M68MAC
 5 :M68MAC SWAP 6 BYTES 20 16 48 40 2C 80 ;M68MAC
 6 :M68MAC 2SWAP A BYTES 20 16 2C AE 00 04 2D 40 00 04 ;M68MAC
 7 :M68MAC OVER 4 BYTES 3D 2E 00 02 ;M68MAC
 8 :M68MAC 2OVER 4 BYTES 2D 2E 00 04 ;M68MAC
 9 :M68MAC >R 2 BYTES 3F 1E ;M68MAC
10 :M68MAC R> 2 BYTES 3D 1F ;M68MAC
11 :M68MAC I 2 BYTES 3D 17 ;M68MAC
12 :M68MAC I' 4 BYTES 3D 2F 00 02 ;M68MAC
13 :M68MAC J 4 BYTES 3D 2F 00 04 ;M68MAC
14 -->
15
```

(Continued on page 102)

Forth Compiler (Listing Continued, text begins on page 68)

Listing Two

```
Screen # 38

 0 ( M68K Cross Compiler -- Comparison operations )
 1 :M68MAC = 6 BYTES 30 1E 32 1E B2 40
 2           8 BYTES 57 C0 02 40 00 01 3D 00 ;M68MAC
 3 :M68MAC < 6 BYTES 30 1E 32 1E B2 40
 4           8 BYTES 5D C0 02 40 00 01 3D 00 ;M68MAC
 5 :M68MAC > 6 BYTES 30 1E 32 1E B2 40
 6           8 BYTES 5E C0 02 40 00 01 3D 00 ;M68MAC
 7 :M68MAC MIN 6 BYTES 30 1E 32 16 B0 41
 8             6 BYTES 6F 02 C1 41 3C 80 ;M68MAC
 9 :M68MAC MAX 6 BYTES 30 1E 32 16 B0 41
10             6 BYTES 6C 02 C1 41 3C 80 ;M68MAC
11 -->
12
13
14
15

Screen # 39

 0 ( M68K Cross Compiler -- Comparison operations )
 1 :M68MAC D= 6 BYTES 20 1E 22 1E B2 80
 2            8 BYTES 57 C0 02 40 00 01 3D 00 ;M68MAC
 3 :M68MAC D< 6 BYTES 20 1E 22 1E B2 80
 4            8 BYTES 5D C0 02 40 00 01 3D 00 ;M68MAC
 5 :M68MAC D> 6 BYTES 20 1E 22 1E B2 80
 6            8 BYTES 5E C0 02 40 00 01 3D 00 ;M68MAC
 7 -->
 8
 9
10
11
12
13
14
15

Screen # 40

 0 ( M68K Cross Compiler -- Comparison operations )
 1 :M68MAC 0= 2 BYTES 4A 5E
 2            8 BYTES 57 C0 02 40 00 01 3D 00 ;M68MAC
 3 :M68MAC NOT 0= ;M68MAC
 4 :M68MAC 0< 2 BYTES 4A 5E
 5            8 BYTES 5D C0 02 40 00 01 3D 00 ;M68MAC
 6 :M68MAC 0> 2 BYTES 4A 5E
 7            8 BYTES 5E C0 02 40 00 01 3D 00 ;M68MAC
 8 :M68MAC D0= 2 BYTES 4A 9E
 9             8 BYTES 57 C0 02 40 00 01 3D 00 ;M68MAC
10 :M68MAC D0< 2 BYTES 4A 9E
11             8 BYTES 5D C0 02 40 00 01 3D 00 ;M68MAC
12 :M68MAC D0> 2 BYTES 4A 9E
13             8 BYTES 5E C0 02 40 00 01 3D 00 ;M68MAC
14 -->
15

Screen # 41

 0 ( M68K Cross Compiler -- Comparison operations )
 1 :M68MAC AND 4 BYTES 30 1E C1 56 ;M68MAC
 2 :M68MAC OR  4 BYTES 30 1E 81 56 ;M68MAC
 3 :M68MAC XOR 4 BYTES 30 1E B1 56 ;M68MAC
 4 :M68MAC 1'S 2 BYTES 46 56 ;M68MAC
 5 -->
 6
 7
 8
```

```
 9
10
11
12
13
14
15

Screen # 42

 0 ( M68K Cross Compiler -- Memory and I/O operations )
 1 :M68MAC ! 6 BYTES 30 1E 3B 9E 00 00 ;M68MAC
 2 :M68MAC @ 6 BYTES 30 16 3C B5 00 00 ;M68MAC
 3 :M68MAC 2! 6 BYTES 30 1E 2B 9E 00 00 ;M68MAC
 4 :M68MAC 2@ 6 BYTES 30 1E 2D 35 00 00 ;M68MAC
 5 :M68MAC +! 8 BYTES 30 1E 32 1E D3 75 00 00 ;M68MAC
 6 :M68MAC C! 8 BYTES 30 1E 32 1E 1B 81 00 00 ;M68MAC
 7 :M68MAC C@ A BYTES 30 16 42 41 12 35 00 00 3C 81 ;M68MAC
 8 :M68MAC FILL 8 BYTES 30 1E 32 1E 30 5E D1 CD
 9              8 BYTES 60 02 10 C0 51 C9 FF FC ;M68MAC
10 -->
11
12
13
14
15

Screen # 43

 0 ( M68K Cross Compiler -- Memory and I/O operations )
 1 :M68MAC AW! 4 BYTES 30 5E 30 9E ;M68MAC
 2 :M68MAC AW@ 4 BYTES 30 56 3C 90 ;M68MAC
 3 :M68MAC AL! 4 BYTES 20 5E 30 9E ;M68MAC
 4 :M68MAC AL@ 4 BYTES 20 5E 3D 10 ;M68MAC
 5 :M68MAC CAW! 6 BYTES 30 5E 30 1E 10 80 ;M68MAC
 6 :M68MAC CAW@ 8 BYTES 30 56 42 40 10 10 3C 80 ;M68MAC
 7 :M68MAC CAL! 6 BYTES 20 5E 30 1E 10 80 ;M68MAC
 8 :M68MAC CAL@ 8 BYTES 20 5E 42 40 10 10 3D 00 ;M68MAC
 9 :M68MAC 2AW! 4 BYTES 30 5E 20 9E ;M68MAC
10 :M68MAC 2AW@ 4 BYTES 30 5E 2D 10 ;M68MAC
11 :M68MAC 2AL! 4 BYTES 20 5E 20 9E ;M68MAC
12 :M68MAC 2AL@ 4 BYTES 20 56 2C 90 ;M68MAC
13 :M68MAC AFILL 8 BYTES 30 1E 32 1E 20 5E 60 02
14                6 BYTES 10 C0 51 C9 FF FC ;M68MAC
15 ;S

Screen # 44

 0 ( Definitions required for FORTH-79 )
 1 : <BUILDS CREATE SMUDGE 0 , ;
 2 : ENDIF [COMPILE] THEN ; IMMEDIATE
 3 ;S
 4 Note.. the above definitions work on my system which is a
 5 combination fig FORTH and FORTH-79.  However, the definition
 6 of <BUILDS may not work with a true FORTH-79 system.  The
 7 compatability depends on how the word DOES> operates in your
 8 system.  For the system described in Leo Brodie's book Starting
 9 FORTH the definition would be:
10 : <BUILDS CREATE ;
11 You will need to write an appropriate definition for your
12 system, the ones given above should serve as guides.
13
14
15
```

End Listing Two

(Continued on next page)

Forth Compiler (Listing Continued, text begins on page 68)

Listing Three

```
Screen # 8

 0 ( Eratosthenes Sieve Prime Number program in FORTH )
 1 ( by Jim Gilbreath, BYTE September 1981 page 190 )
 2 FORTH DEFINITIONS DECIMAL
 3 8190 CONSTANT SIZE      0 VARIABLE FLAGS        SIZE ALLOT
 4
 5 : DO-PRIME      FLAGS SIZE 1 FILL
 6                 0 SIZE 0
 7                 DO FLAGS I + C@
 8                    IF I DUP + 3 + DUP I +
 9                         BEGIN DUP SIZE <
10                         WHILE 0 OVER FLAGS + C! OVER + REPEAT
11                         DROP DROP 1+
12                    THEN
13                 LOOP
14                 .  ." primes " ;
15 ;S

Screen # 9

 0 ( Eratosthenes Sieve Prime Number program for M68K compiler )
 1 ( Original by Jim Gilbreath, BYTE September 1981 page 190 )
 2 DECIMAL 0 M68CON #0 ( Note.. 0 is used a lot in the following )
 3 8190 CONSTANT SIZE      SIZE M68CON SIZE ( Both forms needed )
 4 M68VAR FLAGS    SIZE M68ALLOT
 5
 6 :M68MAC DO-PRIME FLAGS SIZE 1 LITERAL FILL
 7                 #0 SIZE #0
 8                 DO FLAGS I + C@
 9                    IF I DUP + 3 LITERAL + DUP I +
10                         BEGIN DUP SIZE <
11                         WHILE #0 OVER FLAGS + C! OVER + REPEAT
12                         DROP DROP 1+
13                    THEN
14                 LOOP ;M68MAC
15 -->

Screen # 10

 0 ( Test program to run the prime number program )
 1 HEX 7000. M68INIT ( Open the output file )
 2 :M68MAC INIT ( Initialize all the registers )
 3         800.    A5LD    ( Load variable pointer )
 4         4000.   A6LD    ( Load data stack pointer )
 5         7800.   A7LD    ( Load return stack pointer )
 6 ;M68MAC
 7 DECIMAL
 8 :M68K TEST ( Run the prime number test ten times )
 9         INIT
10         10 LITERAL #0 DO DO-PRIME DROP LOOP
11 ;M68K
12 M68END ( Close the output file )
13 ;S
14
15


Screen # 11

 0 ( Eratosthenes Sieve Prime Number program improved version )
 1 ( Original by Jim Gilbreath, BYTE September 1981 page 190 )
 2 DECIMAL 0 M68CON #0 ( Note.. 0 is used a lot in the following )
 3 8190 CONSTANT SIZE      SIZE M68CON SIZE ( Both forms needed )
 4 SIZE M68CARY FLAGS
 5
 6 :M68MAC DO-PRIME #0 FLAGS SIZE 1 LITERAL FILL
```

(Continued on page 106)

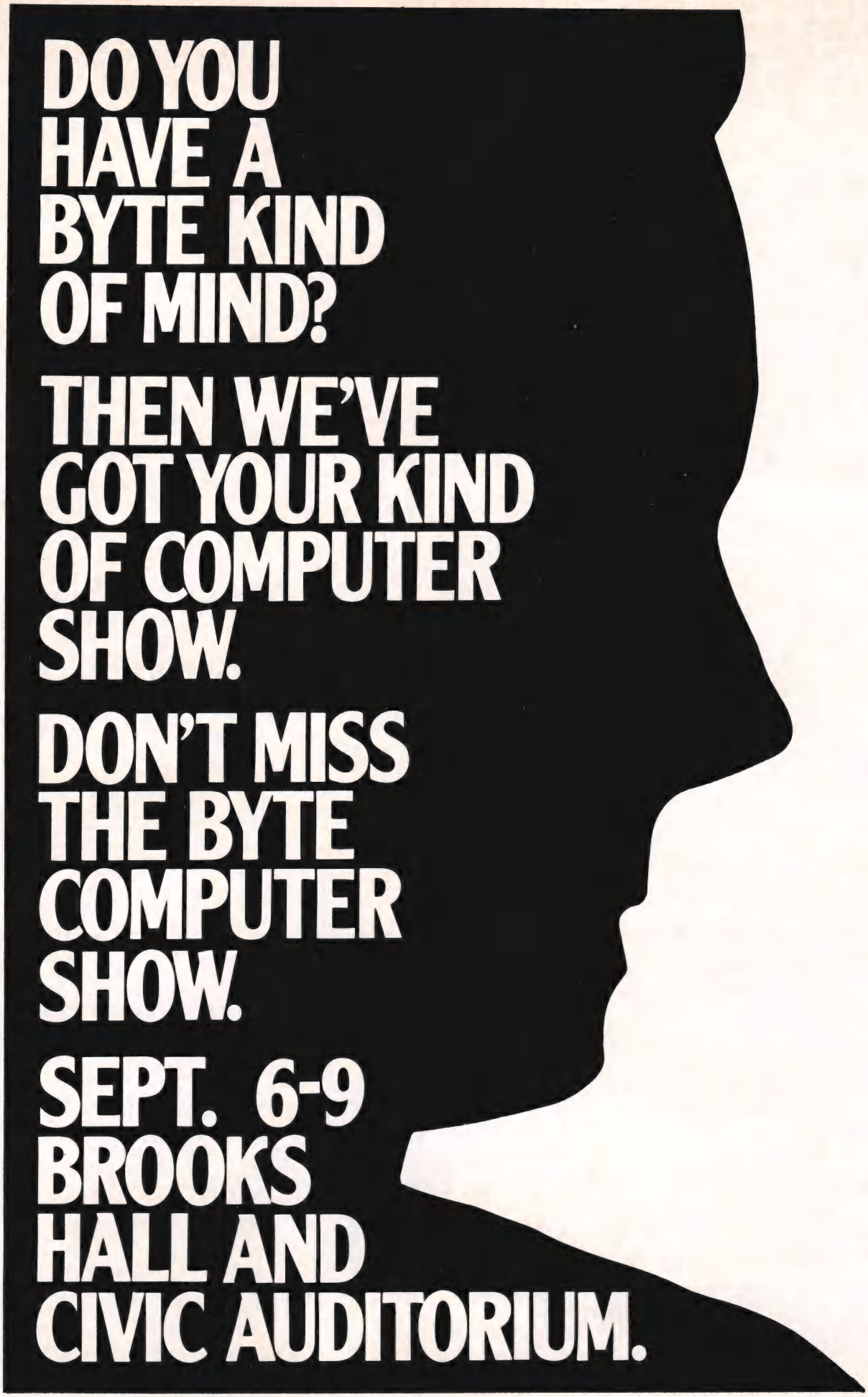
Forth Compiler (Listing Continued, text begins on page 68)

Listing Three

```
 7                  #0 SIZE #0
 8                  DO I FLAGS C@
 9                     IF I 2* 3 LITERAL + DUP I +
10                          BEGIN DUP SIZE <
11                          WHILE #0 OVER FLAGS C! OVER + REPEAT
12                          2DROP 1+
13                     THEN
14                  LOOP ;M68MAC
15 -->

Screen # 12

 0 ( Test program to run the prime number program )
 1 HEX 7000. M68INIT ( Open the output file )
 2 :M68MAC INIT ( Initialize all the registers )
 3          800.     A5LD     ( Load variable pointer )
 4          4000.    A6LD     ( Load data stack pointer )
 5          7800.    A7LD     ( Load return stack pointer )
 6 ;M68MAC
 7 DECIMAL
 8 :M68K TEST ( Run the prime number test ten times )
 9          INIT
10          10 LITERAL #0 DO DO-PRIME DROP LOOP
11 ;M68K
12 M68END ( Close the output file )
13 ;S
14
15
        .PROC   PRIME
;
;  Eratosthenes Sieve Prime Number program in M68000 assembly language.
;  This program provides a baseline for evaluating the performance of the
;  M68K compiler.
;
;  Register variables:
;       D0.. Temporary storage
;       D1.. Number of iterations
;       D2.. I - DO loop counter
;       D3.. P - candidate prime number
;       D4.. K - array index used with P
;       D5.. COUNT - number of primes
;       D6.. SIZE - size of the flags array
;
;       A0.. FLAGS - base address of the FLAGS array
;       A1.. Temporary address register used for initializing FLAGS
;
;
;  Note..
;       This program does not correspond exactly to the FORTH version but
;       the algorithm is the same so this should be a fair comparison.
;       The portions of the FORTH code which correspond to the sections of
;       assembler code are indicated in the comments.
;
;
SIZE    .EQU    8190
FLAGS   .EQU    800H            ;Base address of the FLAGS array
ITER    .EQU    10              ;Number of iterations of the sieve
;
        MOVE.W  #ITER,D1
        MOVE.W  #SIZE,D6
        MOVEA.W #FLAGS,A0
        BRA.S   ENDIL           ;Enter the iteration loop at the proper place
STARTIL                         ;Start of the iteration loop
;
;  FORTH code:
;       FLAGS SIZE 1 FILL
;
MOVEQ   #1,D0
MOVE.W  D6,D2           ;Load SIZE into D2
```

```
        MOVEA.L A0,A1           ;Address of element of FLAGS to set
        BRA.S   $02
$01     MOVE.B  D0,(A1)+
$02     DBRA    D2,$01
;
;  FORTH code:
;       0 SIZE 0 DO
;
        CLR.W   D5              ;Clear prime counter
        CLR.W   D2              ;Clear DO loop counter
DOLOOP
;
;  FORTH code:
;       FLAGS I + C@ IF
;
        BTST    #0,0(A0,D2.W)
        BEQ.S   THEN            ;If false, skip the true part of IF structure
;
;  FORTH code:
;       I DUP + 3 + DUP I +
;
        MOVE.W  D2,D3
        ADD.W   D3,D3
        ADDQ.W  #3,D3           ;P=I+I+3
        MOVE.W  D3,D4
        ADD.W   D2,D4           ;K=P+I
;
;  FORTH code:
;       BEGIN DUP SIZE <
;       WHILE 0 OVER FLAGS + C! OVER + REPEAT
;       DROP DROP 1+
;
BEGIN   CMP.W   D6,D4
        BGE.S   $03
        CLR.B   0(A0,D4.W)
        ADD.W   D3,D4
        BRA.S   BEGIN
$03     ADDQ.W  #1,D5           ;Update prime counter
;
;  FORTH code:
;          THEN
;       LOOP
;
THEN    ADDQ.W  #1,D2
        CMP.W   D6,D2
        BLT.S   DOLOOP
;
;
ENDIL   DBRA    D1,STARTIL      ;Repeat for requested number of iterations
        .END
```

End Listings


DO YOU
HAVE A
BYTE KIND
OF MIND?
THEN WE'VE
GOT YOUR KIND
OF COMPUTER
SHOW.
DON'T MISS
THE BYTE
COMPUTER
SHOW.
SEPT. 6-9
BROOKS
HALL AND
CIVIC AUDITORIUM.

THE BYTE COMPUTER SHOW/San Francisco '84
a unique and informative 25-session conference.

SCHEDULE BY GROUP

KEYNOTE	DAY	TIME
KN-1 Bit-Pusher Perspectives	9/6	11:00-12:30
HARDWARE HELPERS		
HH-1 Who Needs 32 Bits?	9/7	11:00-12:30
HH-2 Is PC Compatibility Holding Us Back?	9/7	5:00- 6:30
HH-3 Adding-On For A Supercharged System	9/8	2:00- 3:30
HH-4 The 1200 bps Modem: Users Report	9/8	5:00- 6:30
SOFTWARE SAVINGS		
SS-1 User Agreements: A New Day Dawning?	9/6	2:00- 3:30
SS-2 Programming Environments: New Tools and Techniques	9/7	5:00- 6:30
SS-3 The Home-Brew Data Base: Tips for Home Brewers	9/8	5:00- 6:30
LANGUAGE LABORATORY		
LL-1 Micro Language Forum	9/6	5:00- 6:30
LL-2 C Language Tradeoffs	9/7	2:00- 3:30
LL-3 BASIC: Can It be Saved?	9/8	11:00-12:30

APPLICATIONS FRONTIER	DAY	TIME
AF-1 Home/Family Management: Beyond the Recipe Collection	9/6	2:00- 3:30
AF-2 Your Personal Robot	9/7	11:00-12:30
AF-3 Systems for the Handicapped	9/7	2:00- 3:30
AF-4 When Less is More: Notebook Computers	9/8	11:00-12:30
SOFTWARE HORIZONS		
SH-1 Next Generation OS: Are Icons Inevitable?	9/6	5:00- 6:30
SH-2 Beyond Words: Idea Processing	9/7	11:00-12:30
SH-3 AI Gateways to Natural Languages	9/7	2:00- 3:30
SH-4 Voice Pattern Recognition	9/8	2:00- 3:30
GRAPHICS GALORE		
GG-1 Keyboard Alternatives	9/6	5:00- 6:30
GG-2 Low Bucks Graphics Add-Ons	9/7	5:00- 6:30
GG-3 Micro Graphics Applications	9/8	11:00-12:30
THE BEST IS YET TO COME		
YC-1 Coming Attractions: The Computer/Video Interface	9/6	2:00- 3:30
YC-2 Japanese Computer Trends	9/8	2:00- 3:30
YC-3 Mass Storage Alternatives	9/8	5:00- 6:30

The FVG Standard Floating-Point Extension

by Ray Duncan
and Martin Tracy

[*Forth's development has been rather controversial, with even the nature of the language tending to work against standardization efforts. Perhaps because of the original fixed-point orientation of Forth, the movement to standardized floating-point operations for the language has lagged behind some of the standardization efforts of other high-level languages. The Forth Vendors Group, an association of vendors of Forth systems, has recently adopted its own standard for a floating-point extension to Forth. We present it here so that readers may examine what they may well be seeing in many of the Forth packages distributed in the future. If a large percentage of vendors choose to implement this, it may well have a significant impact in the next round of standardization by the Forth Standards Team. At the very least, it provides a look at one of the more recent attempts to bring more uniformity and portability to the language. — Ed.*]

In its recent release of the Forth-83 Standard, the Forth Standards Team concerned itself only with the core of the language. No attempt was made to introduce standardization in such areas as floating point, the interface to host operating systems, graphics, or file management. The Forth Vendors Group, hoping to help Forth find broader usage and acceptance, decided to take the initiative by creating and adopting the "FVG Standard Floating Point Extension." The document was written by Martin Tracy of Micromotion and Ray Duncan of Laboratory Microsystems, based on input from several vendors including Talbot Microsystems, Rising Star, Dysan, Miller Microsystems, and Creative Solutions.

The FVG Standard Floating Point Extension is intended to serve chiefly as a guide for implementors, and is a minimum functional description including command and function names and a few new data types. Although this FVG Standard Extension is not binding on any of the FVG members, we expect that most of the major vendors will be modifying their existing floating-point packages to conform with it. It will be submitted to the next meeting of the Forth Standards Team (expected to be in 1987) for consideration as part of the next Forth Standard. If Forth application programs that perform floating-point operations follow the guidelines laid out in the FVG Standard Floating Point Extension, referencing only names and functions described in that document, then such programs should then be portable without changes to the various floating-point implementations created by the different vendors.

"The FVG Standard is intended to serve chiefly as a guide for implementors, and is a minimum functional description . . ."

Ray Duncan, Laboratory Microsystems, P.O. Box 10430, Marina Del Rey, CA 90295.
Martin Tracy, Micromotion, 12077 Wilshire Blvd., Suite 506, Los Angeles, CA 90025.

The Forth Vendors Group Standard Floating-Point Extension

Syntax of Floating Point Numbers

If the floating-point extension word set has been overlaid onto an 83-Standard Forth system, a string of the following form will be compiled or interpreted as a real number:

nnnn.nnnExx

The "E" signifier is mandatory to force the conversion of a real number. The presence of numeric digits before or after the "E" is not required by this specification but may be mandatory in some implementations. A "–" sign may precede both the mantissa and the exponent, a leading "+" sign is also permissible on the exponent. A decimal point is optional and may occur anywhere in the mantissa. For example, all of the following numbers are legal:

```
-.0001E5
100.0E+0
1000.E-15
```

The word FNUMBER may be used by application programs to convert strings into real numbers. The current system BASE is assumed to be DECIMAL whenever floating-point numbers are being converted or displayed. If it is not, the results are undefined.

Display of Floating-Point Numbers

Two primitive display operations are always available: **F.**, which adjusts the position of the decimal point and prints the number without an exponent, and **E.**, which prints the number in exponential notation. More complex output formatting capabilities may be present in some systems and are described in the Optional Floating-Point Words glossary.

Floating-Point Stacks

Floating-point implementations differ in that some have a separate floating-point stack while others keep all integer and real numbers on the same parameter stack. This proposed extension does not concern itself with the relative benefits or disadvantages of either type. The source or destination of a real-number argument is always the floating-point stack, if separate stacks exist.

Portability

To ensure portability of an application, it is important that the programmer *not* take advantage of his knowledge of the physical format of real numbers. The floating-point variants of the memory access and stack operators should be used in all cases for real numbers, even when they happen to coincide with other Forth operators. For example, if the system supports a 32-bit real-number format, FDUP and 2DUP would have the same effect—but FDUP should be used for the sake of portability.

Floating-Point Glossary

The floating-point extension word set is described below in the standard glossary format. The symbols indicate the order in which input parameters have been placed on the stack. Three dashes ("---") denote the execution point; any parameters left on the stack are listed. In this notation, the top of the stack is to the right. Symbols include:

addr	memory address
b	8-bit byte (upper 8 bits zero or ignored)
r,r1	floating-point number (length is implementation dependent)
f	boolean flag, 0=false, –1=true
n	16-bit, signed integer
u	16-bit, unsigned integer
d	32-bit, signed integer
du	32-bit, unsigned integer

Required Floating-Point Arithmetic Operators

F+ r1 r2 --- r3

Floating-point addition, leave the sum of r1 plus r2.

F– r1 r2 --- r3

Floating-point subtraction, leave the difference of r1 minus r2.

F* r1 r2 --- r3

Floating-point multiplication, leave the product of r1 and r2.

F/ r1 r2 --- r3
Floating-point division, leave the quotient of r1 divided by r2.

F** r1 r2 --- r3
Leave the value of r1 raised to the power r2, i.e. $r1^{r2}$.

FABS r --- [r]
Leave the absolute value of real number.

FNEGATE r --- –r
Change the sign of a real number.

FSQRT r1 --- r2
Floating-point square root.

FMAX r1 r2 --- rmax
Leave the larger of two real numbers.

FMIN r1 r2 --- rmin
Leave the smaller of two real numbers.

Required Floating-Point Transcendental Functions

FLOG r1 --- r2
Log to the base 10.

FLN r1 --- r2
Log to the base e.

FALOG r1 --- r2
Leave the value of 10 raised to the power of r1.

FALN r1 --- r2
Leave the value of e raised to the power of r1.

FSIN r1 --- r2
Leave the sine of r1. Input argument is in radians.

FCOS r1 --- r2
Leave the cosine of r1. Input argument is in radians.

FTAN r1 --- r2
Leave the tangent of r1. The tangent of $\pi/2$ or $3\pi/2$ radians returns the largest real number representable in the implementation's binary format.

FASIN r1 --- r2
Arc-sine, valid for
$-1 <= r1 <= 1$.
Returns result in range $-\pi/2$ to $\pi/2$ radians.

FACOS r1 --- r2
Arc-cosine, valid for $-1 <= r1 <= 1$.
Returns result in range 0 to π radians.

FATAN r1 --- r2
Arc-tangent, valid for all real r1. Leaves result in range $-\pi/2$ to $\pi/2$ radians.

Required Floating-Point Logical Operators

The logical operators will regard two real numbers as equal if they differ only by a small amount. This "fuzz factor" is related to the magnitude of the real numbers and is implementation dependent.

F0= r --- f
True if floating-point number is equal to zero. The real number is removed from the floating-point stack, and the flag is left on top of the Forth parameter stack.

F0< r --- f
True if floating-point number is less than zero. The real number is removed from the floating-point

stack, and the flag is left on top of the Forth parameter stack.

F= r1 r2 --- f

True if floating-point number r1 is equal to floating-point number r2. The real numbers are removed from the floating-point stack, and the flag is left on top of the Forth parameter stack.

F< r1 r2 --- f

True if floating-point number r1 is less than floating-point number r2. The real numbers are removed from the floating-point stack, and the flag is left on top of the Forth parameter stack.

Required Floating-Point Stack Operators

FDROP r ---

Discard floating-point number on top of stack.

FDUP r --- r r

Duplicate floating-point number on top of stack.

FOVER r1 r2 --- r1 r2 r1

Copy second floating-point number on stack to top of stack.

FSWAP r1 r2 --- r2 r1

Interchange two floating-point numbers on top of stack.

FROT r1 r2 r3 --- r2 r3 r1

Bring the third floating-point number to top of stack.

Required Real-Number Handling

FLOAT d --- r

Convert a signed double integer into its real-number equivalent, removing the double integer from the Forth parameter stack and leaving the result on the floating-point stack.

FIX r --- d

Convert a floating-point number to the nearest signed double integer equivalent, removing the real number from the floating-point stack and leaving the double integer result on the Forth parameter stack. The direction of rounding when a real number lies exactly halfway between two integers is implementation dependent. Underflow gives a zero result, overflow is an error condition and the value of the resulting double integer is undefined. Example: 2.7E0 FIX will return the double integer 3.

INT r --- d

Truncate a floating-point number to a signed double integer (round it toward zero), removing the real number from the floating-point stack and leaving the result on the Forth parameter stack. Underflow gives a zero result, overflow is an error condition. Example: 2.7E0 INT will return the double integer 2.

F! r addr ---

Store a floating-point number from the floating-point stack, at the address that is on the top of the Forth parameter stack.

F@ addr --- r

Fetch a floating-point number to the top of the floating-point stack, from the address that is on top of the Forth parameter stack.

FCONSTANT r --- (compilation)
--- r (execution)

A defining word used in the form:

r FCONSTANT cccc

When FCONSTANT is executed, it creates the definition cccc with its parameter field initialized to the value r. When cccc is later executed, the value r is left on top of the floating-point stack.

FVARIABLE --- (compilation)
--- addr (execution)

A defining word used in the form:

FVARIABLE cccc

When FVARIABLE is executed, it creates the definition cccc with its parameter field explicitly uninitialized. When cccc is later executed, the address of the parameter field is left on the stack, so that a F@ or F! operation may access this location.

Required Floating-Point Number Display

E. r ---

Display r in exponential form. The mantissa contains the maximum number of significant digits allowed by the floating-point data format, and the exponent is explicitly displayed even if it is zero. A trailing blank follows. If the current system base is not decimal, an error condition exists. For example, in the Micromotion implementation 12345.67E2 E. will display as .1234567E-07b (where each "b" character denotes an ASCII blank).

F. r ---

Display r on the currently selected output device in fixed-point form; i.e., the location of the decimal point is adjusted as necessary so that no exponent need be displayed. The number of digits specified by the most recent execution of the word PLACES (see below) are printed to the right of the decimal point. A trailing blank follows. For example, 4 PLACES 1.2345E02 F. will display as 123.4500b (where each "b" character denotes an ASCII blank).

PLACES n ---

Set the default number of digits that will be printed to the right of the decimal point by the F. operator.

Optional Floating-Point Commands

The Optional words have the same relationship to the proposed Floating-Point Extension as Controlled Reference Words have to the core 83-Standard; that is, if the function is available in the system, it should have the name and stack effect described below.

Optional Floating-Point Transcendental Operations

FSINH r1 --- r2

Hyperbolic sine.

FCOSH r1 --- r2

Hyperbolic cosine.

FTANH r1 --- r2

Hyperbolic tangent.

FASINH r1 --- r2

Inverse hyperbolic sine.

FACOSH r1 --- r2

Inverse hyperbolic cosine.

FATANH r1 --- r2

Inverse hyperbolic tangent.

PI --- r

Leave the real number π on the stack, represented with the maximum precision possible in the implementation's binary floating-point format.

Optional Floating-Point Logical Operators

F0> r --- f

"True" if the floating-point number is greater than zero. The real number is removed from the floating-point stack, and the flag is left on top of the Forth parameter stack.

F> r1 r2 --- f

"True" if the floating-point number r1 is greater than the floating-point number r2. The real numbers are removed from the floating-point stack, and the flag is left on top of the Forth parameter stack.

Optional Floating-Point Display Words

Suggested error handling for floating-point display: if an overflow occurs during conversion for output, fill the display field with asterisks (the "*" character). If an underflow occurs during conversion, fill the display field

play. Implementation dependent.

E.R r n1 n2 ---
Display r on the currently selected output device in exponential form with n1 digits to the right of the decimal point, right justified in a field of width n2. If the current system base is not decimal, an error condition exists. For example, 1.234E0 5 12 E.R will display as bb.12340E-01 (where each "b" character denotes an ASCII blank).

F.R r n1 n2 ---
Display r on the currently selected output device in fixed-point form with n1 digits to the right of the decimal place, right justified in a field of width n2. Numbers that cannot be represented within the given field width are printed in exponent form. If the current system base is not decimal, an error condition exists. For example, 1.2345E2 4 12 F.R will display as bbbb123.4500 (where each "b" character denotes an ASCII blank).

Optional Miscellaneous Real-Number Handling

FNUMBER addr --- r
Addr points to a counted string that is converted to a real number. The result is left on the floating-point stack. If the syntax of the string is not correct for a floating-point number, an error condition exists and the value of the result is undefined.

PACK d n --- r
Collapse a signed double-integer mantissa and a signed single-integer power of 10 into the corresponding real-number data format. Implementation dependent.

UNPACK r --- d n
Convert a real-number into a signed double integer corresponding to the mantissa and a signed single-integer power of 10. Implementation dependent.

F#BYTES --- n
Constant leaving the number of bytes in a real number as it is represented on the floating-point stack. Implementation dependent.

F#PLACES --- n
Constant leaving the maximum number of significant digits when a real number is converted to its decimal ASCII equivalent for display. Implementation dependent.

Optional Error Handling for Real-Number Operations

At minimum, the following error indicators should be available for inspection by the application program:

- overflow
- underflow
- division by zero
- output conversion overflow
- output conversion underflow
- attempted logarithm of zero
- attempted logarithm of number less than zero
- attempted square root of a number less than zero
- arcsin/arccos of argument outside the legal domain

FPSTAT --- addr
Addr points to an array of bits that contains error indicators for real-number operations. The number of flag bits and their significance is implementation dependent.

FPERR r1 n --- r2
Should be a deferred or vectored word to allow installation of altered error handling by the user. r1 is the result of a real-number operation and n indicates an error type. Sets an appropriate flag in FPSTAT and returns the real number r2, which may or may not be the same as r1, depending on the implementation and type of error.

Micromotion has written a Forth-83 implementation of this FVG Standard Floating-Point Extension. Machine code versions for the 6502, 8086, and 68000 are currently available. Micromotion may be contacted at 12077 Wilshire Blvd., Suite 506, Los Angeles, CA 90025.

DDJ

Reader Ballot
Vote for your favorite feature/article.
Circle Reader Service **No. 196.**

by Anthony Skjellum

Past columns have been devoted to discussions about C and Unix as they are implemented in the real world. In this column, I depart from this precedent by discussing some proposals for increasing the power and flexibility of C. We work in an evolving field. Consequently, many languages go through regular upgrades and improvements (e.g., Fortran IV, Fortran 66 and Fortran 77). C has been more immune to change than other popular languages. The lack of upgrades to C is probably due mainly to its lack of intrinsic functions, but clearly points to the basic elegance and power of C.

I am aware of only a few upgrades to C beyond the language definition specified in *The C Programming Language* by Kernighan and Ritchie. These upgrades involved enumerated types and structure assignment. Both of these features were introduced with Unix version 7 and are detailed in the documentation for Unix version 7.

Structure assignment is useful in what follows, but I will not mention type enumeration. Therefore, although we base our definition of C on the Unix 7 version of C, this is not likely to confuse those who have access only to *The C Programming Language*.

Purists may argue that I have no business recommending changes or upgrades to C. Others may argue that many of the suggestions can be implemented via compiler preprocessors or by function calls and need not be part of the language. (I'll discuss this second point later in the column.) In order to head off the criticism that I am "tampering" with the C language, I offer my recommendations in the form of a new language grammar that is based on C but called X. I chose the letter *X* to denote language extensibility, which is the main point of the following proposals.

Language Extensibility

Most languages allow user-defined functions and subroutines, and many newer languages allow user-defined data tyes. Extensible langauges such as Forth and APL allow functions, operators, and data types to be added to the programming environment in a way that makes them equivalent in stature to predefined operations. Although C retains tremendous flexibility by excluding intrinsic functions, it does not allow user-defined types to be treated as easily as ints, longs or floats. Specifically, one cannot extend the definitions of operators such as Addition or Multiplication to new data types created with Typedef. This means that function calls must be used. Although this is a viable approach, it lacks elegance. This concept is illustrated in the following example.

As part of my C program, I need to define a data type called COMPLEX that will function like Fortran's complex data type. This data type is used for handling complex numbers of the form **A + B** where **i** is the imaginary unit and **A** and **B** are real numbers. This is shown with the definition in Figure 1 (page 116).

I will work with several variables of type COMPLEX (e.g., **alpha** and **beta**), which are defined as follows:

```
COMPLEX alpha, beta;
 /* alpha and beta are complex #'s */
```

Up to this point, we have treated the complex data type as we would built-in types. We can also work with pointers to or arrays of COMPLEX. However, to assign, add, multiply, or subtract these COMPLEX variables, we would have to invent subroutines. Subroutines for two representative operations are illustrated in Figure 2 (page 118).

The pseudocode presented with the subroutines in Figure 2 is the most convenient way to specify the operations desired. If the data types had been intrinsic, we could have used similar real C statements in lieu of subroutines. To utilize +, * or other operators with the COMPLEX data type, we must introduce a mechanism for defining these operations.

Operators

How could we specify new operations? For example, how would we define Addition for the complex data type? Figure 3 (page 120) shows the type of definition that could be used to extend Addition to the COMPLEX type.

The keyword **oper** is new: oper indicates that the definition following it is for an operator. The **return** keyword used in function calls also appears with a similar meaning. Because COMPLEX preceeds oper, this example defines an operation over the COMPLEX data type. Because there are two arguments (alpha, beta), the operator is binary. Finally, note that the + sign is enclosed in graven accents. Quoting by graven accents is chosen as a way to distinguish operator names. We will see that quota-

```
typedef struct /* complex number type definition */
{
                                        double _creal ; /* real part */
                                        double _cimag ; /* imaginary part */
} COMPLEX
```

Figure 1

tion will not always be needed.

To use this new operator—and assuming that = had also been defined—the following statement could be used:

```
gamma = alpha + beta;
        /* add complex numbers */
```

Note that we have omitted the graven accents. Because the + can be distinguished from keywords or identifiers in this context, quoting is not required.

The operator definition specified above gives the X compiler a means to evaluate the Addition request specified in the example statement. The parser would break this statement down until it could pass an argument garnered from the left and right of the Addition operator, much as it does with intrinsic operators and data types. Whether this results in a subroutine call or in-line code would depend on the compiler's implementation.

More on Operators

Operators are a powerful and useful concept. We needn't limit ourselves to defining standard operations for new types. There is nothing to stop the definition of arbitrary operators. A crude facility already exists for this in C via the parameterized #define statement. However, facility just discussed is more general and more consistent with the syntax of C than the preprocessor #define approach. To encompass the generation of in-line code as provided by #define, we would also offer the inline adjective, which could be used as follows:

```
COMPLEX inline oper
    `-`(alpha, beta)
            /* subtraction inline */
```

This keyword would instruct the compiler to generate in-line code (as opposed to a subroutine call) whenever possible. Its use is analogous to the use of the register adjective: the compiler complies when feasible and ignores the request when it cannot comply.

In some cases, C definitions can be shortened when no ambiguity exists (e.g., "unsigned" instead of "unsigned int"). Therefore, "inline" would replace "inline oper" in actual practice. Furthermore, operators would, by default, work on and return integers, as functions do by default.

Other Uses for Operators

In my view, operators would be used not only to define existing operations over new data types, but also for specifying other operations over new as well as existing data types. These new operators would normally have alphanumeric names and would thus require quoting in graven accents when they appear in expressions. For example, we define the operation of NAND (negated and) for integers as follows (no graven accents are required in the definition, but they are required in the invocation below):

```
int oper nand(a,b)
int a,b;
{
    return(~(a & b));
}
```

To use this in an actual expression, we would have to quote the nand:

```
c = a `nand` b;
```

Operator Hierarchy

C already has a built-in hierarchy for known operations. The most reasonable approach is to give user-defined operators the lowest priority. This might require more parentheses, but seems logical.

Pointers to Operators

C provides the facility to use pointers to functions. It could potentially prove useful to have pointers to operators as well. You specify a function's address by its name without trailing parentheses. Unfortunately, operator names are used in this way to indicate the operation they represent. In order to remove the ambiguity in requesting the pointer, the operator name could be parenthesized—for example; (+) or (`nand`).

Using pointers to operators implies that defined operations must have subroutines associated with them. Thus,

truly in-line operators could have no pointers associated with them.

Dichotomy of Operators and Functions

Functions and operators are almost the same thing. However, the compiler must know if an operator is binary or unary. Therefore, the operator definition must be available before you use it. On the other hand, arguments to C functions are not checked for number or type. Therefore, we choose to keep operators and functions separate, although there is nothing to prevent operators from using function calls.

In order to avoid lexical conflicts, operator and function names would have to be different. This is also desirable from a programming viewpoint, in order to avoid confusion and errors.

```
Assignment: alpha = A + iB; /* pseudo code */
Function:
        calling sequence    (K&R C):        cassign(&alpha,A,B);
        calling sequence    (Unix 7 C):     alpha = cassign(A,B);
        function definition (K&R C):
                cassign(comp,a,b)
                COMPLEX *comp;
                double a,b;
                {
                        comp->_creal = a;
                        comp->_cimag = b;
                }
        function definition (Unix 7 C);
                COMPLEX cassign(a,b)
                double a,b;
                {
                        COMPLEX temp; /* temporary variable */
                        temp._creal = a;
                        temp._cimag = b;
                        return(temp); /* return structure */
                        }
Addition: gamma = alpha + beta; /* pseudo code */
Function:
        calling sequence    (K&R C):        cadd(&gamma,&alpha,&beta);
        calling sequence    (Unix 7 C):     gamma = cadd(alpha,beta);
        function definition (K&R C):
                cadd(gamma,alpha,beta)
                COMPLEX *gamma; /* destination */
                COMPLEX *alpha; /* addend */
                COMPLEX *beta; /* augend */
                {
                        gamma->_creal = alpha->_creal + beta->_creal
                        gamma->_cimag = alpha->_cimag + beta->_cimag
                }
        function definition (Unix 7 C);
                COMPLEX cadd(alpha,beta)
                COMPLEX alpha,beta; /* addend, augend */
                {
                        COMPLEX temp; /* temporary */
                        temp._creal = alpha._creal + beta._creal;
                        temp._cimag = alpha._cimag + beta._cimag;
                        return(temp);
                }
```

Figure 2

Other Proposals

With the addition of operators, the X grammar provides a much more consistent programming environment than standard C. However, there are some other points that deserve consideration. The first of these is the need to provide a means to handle subroutines with a variable number of arguments.

Because C makes no assumptions about its function library, the user is free to write his own library, should the standard functions prove inadequate. However, the user cannot properly handle functions that have variable numbers of arguments, as must be done by printf( ), scanf( ), and their relatives. We solve this problem by introducing a typing adjective called **vec** (vector). This adjective is used to indicate that the number of arguments to the function is variable. For example, the fictitious function my_printf( ), which allows variable arguments (and returns an integer), would be defined as follows:

```
vec int my_printf(argcnt,argvec)
int argcnt;
char *argvec[ ];
{
                /* code goes here */
}
```

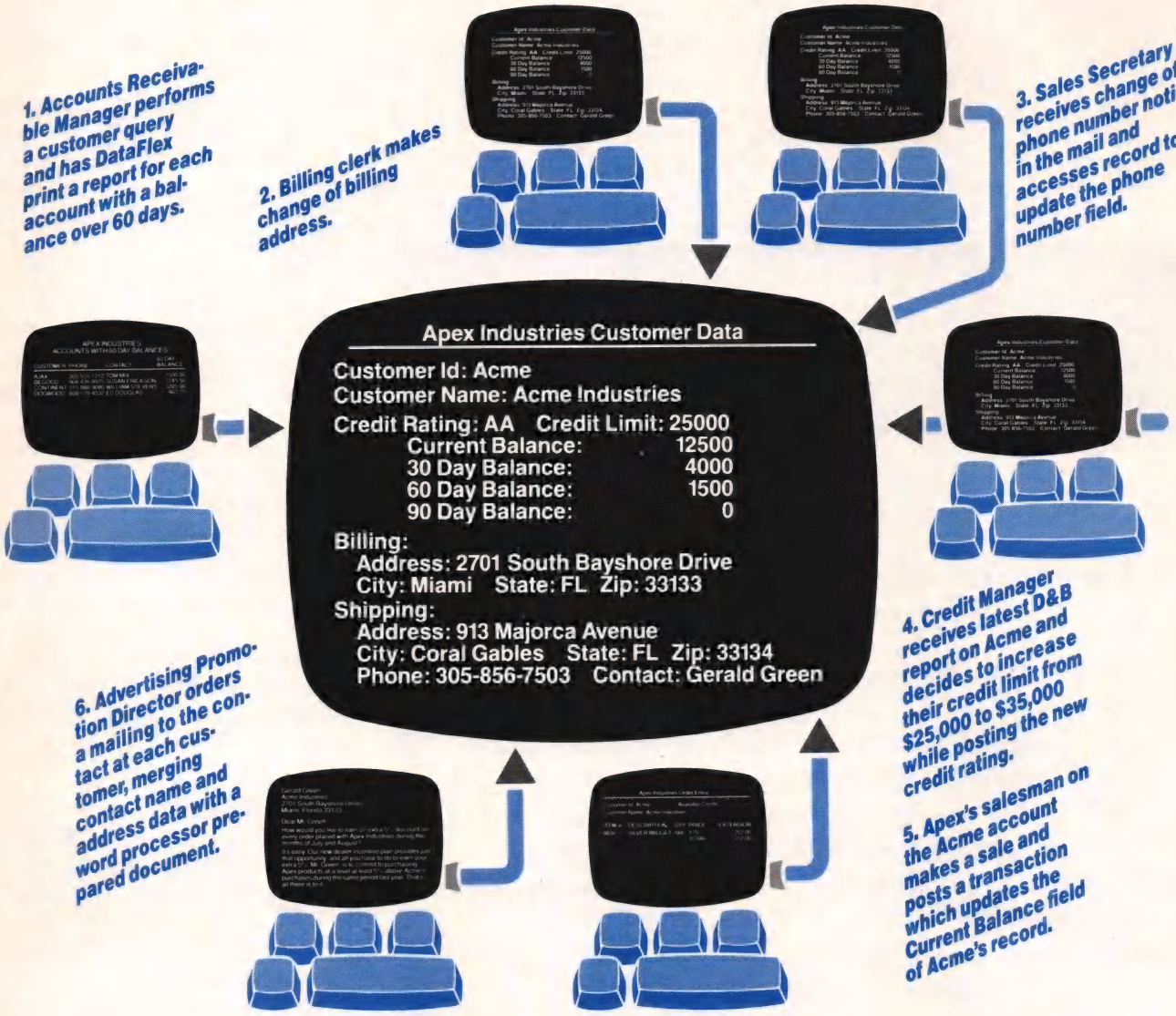
A function declared with vec always has two arguments: argcnt and argvec. These variables are analogous to main( )'s (argc,argv) pair. Before use, a definition of the form:

```
vec my_printf( );
```

would be included in each file where my_print( ) is referenced. This definition causes command line arguments to be processed normally; the rightmost argument is pushed (placed on the stack) first and the leftmost last when code is generated. However, the two additional arguments argcnt and argvec are also stacked. The argvec variable points to the stack location where the first real argument is located. Since normal stacks are push-down, this should provide the arguments in the correct order—argcnt contains the number of arguments plus one to account for argvec. This makes

ALL AT ONCE!

AND NEVER A "LOCKED OUT" USER!



DataFlex is the only application development database which ***automatically*** gives you true multi-user capabilities. Other systems can lock you out of records or entire files for the full time they are being used by someone else. DataFlex, however, locks only the data being changed, and ***only*** during the micro-seconds it takes to actually write it to the file! The updated record is then immediately available. The number of users who can access, and change, records at the same time is limited only by the number of terminals on your system or network. Call or write today for all the details on DataFlex…the true multi-user database.

DATAFLEX™

DATA ACCESS CORPORATION
8525 SW 129 Terrace, Miami, FL 33156 (305) 238-0012
Telex 469021 DATA ACCESS CI

Compatible with CP/M-80, MSDOS networks, MP/M-86, Novell Sharenet, PC-Net, DMS Hi-net, TurboDOS multi-user, Molecular N-Star, Televideo MmmOST, Action DPC/OS, IBM PC w/Corvus, OMNINET, 3Com EtherSeries and Micromation M/NET.
MSDOS is a trademark of Microsoft. CP/M and MP/M are trademarks of Digital Research

Circle **no. 20** on reader service card.

```
COMPLEX oper `+`(alpha,beta) /* X
grammar */
COMPLEX alpha,beta ;
{
    COMPLEX _ _temp; /* temporary
    */
    _ _temp._creal = alpha._creal + beta._creal ;
    _ _temp._cimag = alpha._cimag + beta._cimag ;
    return(_ _temp) ; /* return result */
```

Figure 3

it completely analogous to argc. The argument argvec always contains an address, but this is not very useful if no arguments were specified in the function call.

To illustrate the stacking mechanism, imagine that we invoke my_printf( ) as follows:

```
my_printf( )(arg1,arg3,arg4,arg5) ;
```

For this specific call, the stacking arrangement (excluding any special register saves) woud look as specified in Figure 4. Note that argcnt is six (five arguments) for this case, as described above.

It might be worthwhile to have variable argument calls, even if the function were not declared as using this calling convention. To allow this, we introduce the ellipsis (. . .) concept into the argument string. If my_printf( ) were not declared as vec, we could force a variable argument format as follows:

```
my_printf( )
    (arg1,arg2,arg3,arg4,arg5 . . . );
```

Including the ellipsis mark for this variety of call seems to improve readability, but is not required for compatibility with current C usage.

Fixed Arguments

The argvec variable always points to the first variable specified in the function call. However, the function definition could still explicitly declare a finite number of arguments that it may wish to examine directly. For example, if the first argument of my_printf( ) were a control string, we could declare my_printf( ) as shown in Figure 5.

Notice that the contents of control_string would be meaningless if argcnt were less than two.

One final note about variable argument control: it enhances a function's abiliity to detect incorrect input. With reference to printf( ), Kernighan and Ritchie state: "A warning: printf uses its first argument to decide how many arguments follow and what their types are. It will get confused if there are not enough arguments or if they are the wrong type."

If implemented with the vec arrangement, printf( ) could at least know if it

has been given the right number of arguments. It still would not know if they were of the correct types.

Variable-length Automatic Arrays

Another element of the X grammar is its ability to declare automatic arrays that possess variable length. Because stack displacements are computed at each entry to a block, a computed size allocation is forced. At worst, a memory allocation mechanism must be tied into the compiler. This latter restriction can be serious if C is used in a very-low-level environment, such as in operating system development. So that the use of this feature can be seen readily, we require the use of the var adjective in conjunction with such definitions. For most purposes, the feature is a welcome enhancement. Where it is inappropriate, this feature should be disabled via a compiler switch.

As a general example, we declare a variable-length array in the routine in Figure 6 (page 122).

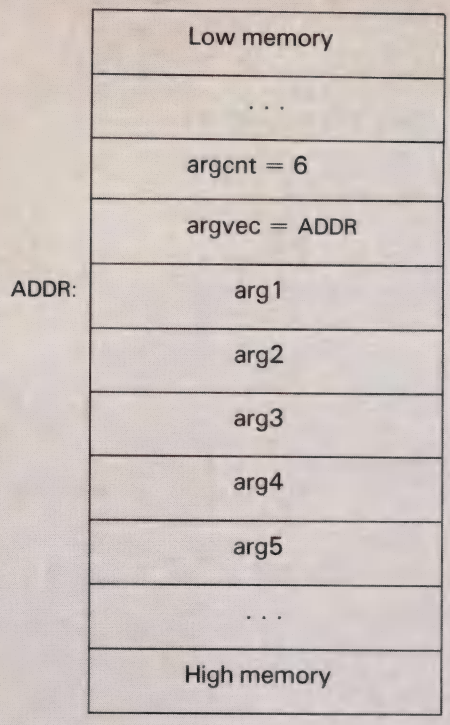
A New Looping Structure

Many loops are unconditional, with breaks generated only from within. Therefore, it is often useful to have an unconditional looping command. This avoids a lot of while(1) sequences. The command could be implemented as follows:

```
    loop
    {
                              . . . code . . .
    }
replaces
    while(1)
    {
                              . . . code . . .
    }
```

Preprocessors and Related Comments

Preprocessors could be used to implement several of the X features mentioned above. The statements and expressions would be expanded by the preprocessor into standard function calls. The preprocessor would also provide subroutines from definitions as needed. New data types could certainly be handled in this way. However, changes to the C parser must be made in order to handle the vec and var features. Trivial additions such as loop can be



Memory Layout for a variable argument function call

Figure 4

```
vec int my_printf(argcnt,argvec,control_string) ;
int argcnt ;
char **argvec ;
char *control_string ;
```

Figure 5

```
/* declare an array of integers one larger than argument */
array_test(length)
int length ;
{
    var int test[length+1]; /* declare array */
    . . .
}
```

Figure 6

handled with the existing C preprocessor.

Some programmers may argue that no additions are needed, because most of the features outlined above can all be achieved through function calls. In my view, the X grammar makes C more (and not less) consistent because it allows intrinsic and user-defined types to be handled similarly. It also allows greater portability by defining a means through which variable argument functions can be handled uniformly. In summation, it turns C into an extensible language while adding only a few new keywords.

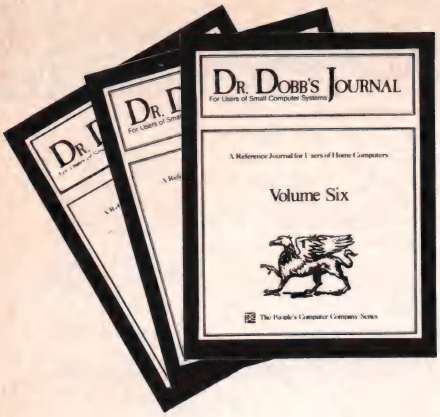
Conclusion

In this column, I have suggested an enhanced C grammar, which was denoted X to indicate extensibility. It is the (Unix 7) C language with enhancements designed to allow the incorporation of user-specified operators into programs. This should provide more flexible and consistent reference to user-defined data types. Also mentioned were variable-length automatic arrays (var) and a mechanism for allowing variable argument functions (vec). Finally, the use of preprocessors for implementing these ideas was mentioned.

I look forward to any other ideas about X or enhanced C that may be forthcoming from readers. **DDJ**

Reader Ballot
Vote for your favorite feature/article.
Circle Reader Service **No. 197.**

ADVERTISER INDEX



Dr. Dobb's Journal

BOUND VOLUMES

Every Issue Available For Your Personal Reference.

Vol. 1 1976

The material brought together in this volume chronicles the development in 1976 of Tiny BASIC as an alternative to the "finger blistering," front-panel, machine-language programming which was then the only way to do things. This is always pertinent for bit crunching and byte saving, language design theory, home-brew computer construction and the technical history of personal computing.

Topics include: Tiny BASIC, the (very) first word on CP/M, Speech Synthesis, Floating Point Routines, Timer Routines, Building an IMSAI, and more.

Vol. 2 1977

1977 found **DDJ** still on the forefront. These issues offer refinements of Tiny BASIC, plus then state-of-the-art utilities, the advent of PILOT for microcomputers and a great deal of material centering around the Intel 8080, including a complete operating system. Products just becoming available for reviews were the H-8, KIM-1, MITS BASIC, Poly Basic, and NIBL. Articles are about Lawrence Livermore Lab's BASIC, Alpha-Micro, String Handling, Cyphers, High Speed Interaction, I/O, Tiny Pilot & Turtle Graphics, many utilities, and even more.

Vol. 3 1978

The microcomputer industry entered its adolescence in 1978. This volume brings together the issues which began dealing with the 6502, with mass-market machines and languages to match. The authors began speaking more in terms of technique, rather than of specific implementations; because of this, they were able to continue laying the groundwork industry would follow. These articles relate very closely to what is generally available today.

Languages covered in depth were SAM76, Pilot, Pascal, and Lisp, in addition to RAM Testers, S-100 Bus Standard Proposal, Disassemblers, Editors, and much, much more.

Vol. 4 1979

This volume heralds a wider interest in telecommunications, in algorithms, and in faster, more powerful utilities and languages. Innovation is still present in every page, and more attention is paid to the best ways to use the processors which have proven longevity—primarily the 8080/Z80, 6502, and 6800. The subject matter is invaluable both as a learning tool and as a frequent source of reference.

Main subjects include: Programming Problems/Solutions, Pascal, Information Network Proposal, Floating Point Arithmetic, 8-bit to 16-bit Conversion, Pseudo-random Sequences, and Interfacing a Micro to a Mainframe—more than ever!

Vol. 5 1980

All the ground-breaking issues from 1980 in one volume! Systems software reached a new level with the advent of CP/M, chronicled herein by Gary Kildall and others (**DDJ's** all-CP/M issue sold out within weeks of publication). Software portability became a topic of greater import, and **DDJ** published Ron Cain's immediately famous Small-C compiler—reprinted here in full!

Contents include: the Evolution of CP/M, and CP/M-Flavored C Interpreter, Ron Cain's C Compiler for the 8080, Further with Tiny BASIC, a Syntax-Oriented Compiler Writing Language, CP/M-to-UCSD Pascal File Conversion, Run-time Library for the Small-C Compiler and, as always, even more!

Vol. 6 1981

Microcomputing was entering a technical maturity in 1981, while continuing to break new ground. This volume includes Dr. Dobb's first all-FORTH issue and the first Dr. Dobb's "Clinic" columns. There is continued coverage of CP/M and Small-C development, along with J.E. Hendrix's Small-VM and Santa Barbara Tiny BASIC for 6809—all here in one giant volume.

Articles include: Pidgin—A Systems Programming Language, The Conference Tree, Write Your Own Compiler with META-4, several exciting Z80 utilties, North Star tidbits, and more!

YES! ☐ Please send me the following Volumes of **Dr. Dobb's Journal.**
☐ ALL 6 for ONLY $125, a savings of over 15%!

Payment must accompany your order.

Please charge my: ☐ Visa ☐ MasterCard
I enclose ☐ Check/money order

Card #_____ Expiration Date_____

Signature_____

Name_____ Address_____

City_____ State_____ Zip_____

Mail to: Dr. Dobb's Journal, 2464 Embarcadero Way, Palo Alto, CA 94303
Allow 6–9 weeks for delivery.

Vol. 1_____ x $23.75 = _____
Vol. 2_____ x $23.75 = _____
Vol. 3_____ x $23.75 = _____
Vol. 4_____ x $23.75 = _____
Vol. 5_____ x $23.75 = _____
Vol. 6_____ x $27.75 = _____
All 6 _____ x $125.00 = _____

Sub-total $_____

Postage & Handling _____
Must be included with order.
Please add $1.25 per book in U.S.
($2.00 each outside U.S.)

TOTAL $_____

Circle **no. 106** on reader service card.

by Michael Wiesenberg

FIG Member Services

The **Forth Interest Group (FIG)** wants you to join up, and its newly elected board is emphasizing member services as one of its prime objectives. The services of this nonprofit, member-supported organization of more than 4800 members and 53 chapters worldwide include publication of *FORTH Dimensions*, the FIG-Tree (on-line computer data base), a job registry, various conferences, user library, membership directory, speaker bureau, catalog of software, educational tapes, and the annual Forth Convention (November 16–17 in Palo Alto, California). You get it all for $15 a year ($27 foreign). **Reader Service No. 101.**

Four Times the Forth

FourByteForth, from Software Architects, is a 32-bit Forth for 68K machines. It runs under CP/M-68K or boots directly on the Sage Computer. Your $250 gets you a full-screen editor, full-screen debugger, interpreter, compiler, decompiler, assembler, and user manual. It's supposed to be fast; the company claims it compiles 125 screens per minute. **Reader Service No. 103.**

More Freeware

I'm highly in favor of programs you get for the cost of a diskette—or free if you supply diskette and postage—and then pay for if you like. Charter Software offers **PC - General Ledger** for the IBM PC and PC compatibles. The program is a cash-receipts journal, check register, journal entry, trial balance, and financial-statement system that also prints checks. If you like it, you send in $50, and then you also get enhancements, updates, phone support, and the manual. **Reader Service No. 113.**

Fortran 77 Library

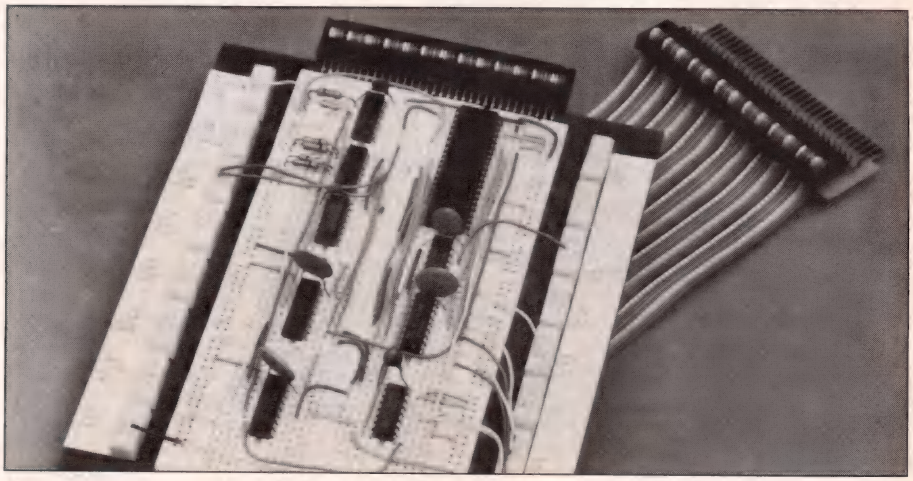
Peerless Engineering Service offers a **Fortran 77 Scientific Subroutine Library** for MSDOS and PCDOS, using Microsoft Fortran 3.13, with 114 scientific subroutines, from statistical to numerical analysis. It includes solutions to third-order differential equations, solutions to *m* equations in *n* unknowns, and various approximations. Need most flavors of logarithms, including operations on complex numbers? Multiple and linear regressions? Fourier analysis? All here. Order directly for $49.95 through the firm's toll-free number. **Reader Service No 119.**

More for C64

Practicorp offers several products of interest to Commodore 64 owners. **PractiFile** is a data base program that handles lists and numbers. The program is designed to handle inventory control, mailing lists, accounts receivable and teachers' grade books; it also monitors charge accounts and bank balances and prints mailing labels or

Breadboard It Yourself

eZ BOARD, from Sabadia Export, provides breadboardng for the masses—a glass epoxy PCB with solderless breadboarding units for building circuits—that allows you to make your own experimental add-ons for personal computers. The product contains four separate distribution buses with 50 tie points each. You get 2090 tie points with a capacity of 24, 14-pin DIP switches. The company says you simply plug in components with lead diameters up to .032 and connect them with ordinary, solid hookup wire. (The firm didn't say .032 *what*. Do you suppose it meant inches? Or maybe .032 angstroms?) Models IPC, APC, and CPC are available for IBM PC, Apple, and Commodore systems (and all hardware-compatible computers of each type), and more are coming. Send $174.95 plus $5 shipping to get board, cable, and connectors. **Reader Service No.111.**



statistical reports. It interacts with other Practicorp software products, plus various word-processing programs. The company claims Practi-File is easy to use, that its ability to handle more than 1000 mailing-list records exceeds the capacity of other DBMS programs for the C64. Fields can be of random size, and maximum record size is 254 characters. Sorts are fast, and you can specify up to five nested keys. You can have multiple lines per record and use arithmetic functions on columns. The disk costs $55. **Reader Service No. 107.**

64 Doctor is a diagnostic program that looks at the system's hardware and pinpoints malfunctions involving keyboard, audio, video, joystick, RS-232 port, disk drive, printer, RAM, or cassette player. It's $29.95 on disk and $24.95 on tape. **Reader Service No. 109.**

Insured Against Infringement

This month's unusual claim is that **IBM PC-compatible ROM BIOS** software from Phoenix is insured against infringement suits. Just think how much better Franklin's financial position would have been had it insured its operating system. If you're an OEM who needs unlimited licensing on this product, $290,000 will do the trick. I like to let *DDJ* readers in on these little bargains. **Reader Service No. 115.**

Word Processing on the C64

Commodore 64 owners sometimes sneer knowingly when someone talks of word processing on the C64; those sceptics cite the machine's limitation of file size to a maximum of about seven pages and its inability to display 80 characters without an 80-column board. Muse Software claims its **Super-Text** addresses those and other problems. For $99, you get unlimited document size, multiple-file search and replace, split-screen abilitiy, 80-column display without additional hardware, on-screen formatting, on-screen help, word wrap, and single-key commands. The product is also available for Atari 400/800/1200XL computers, but without the split-screen feature or 80-column ability. An IBM PC version of Super-Text has just been published. The original version, for the Apple II and IIe, is a perennial bestseller. The Apple and IBM PC versions each cost $175. **Reader Service No. 105.**

Old Fortrans for New Cs

Don't throw away your old Fortran code just because you're updating to a trendy new C environment. **FORTRIX**, from Rapitech Systems, converts Fortran programs and files to maximally portable C code. The new C code retains the essence of the original Fortran source, fully structured and parallel to the original, yet in functionally equivalent C statements. It costs $2500, configured, the firm says, for "various" Unix environments. **Reader Service No. 121.**

T_EX and Tingler! on 3000

T_EXies will be pleased to hear they can now get **T_EX** for their HP-3000s, from **TeXeT**, and produce flawless, typeset-quality documents. They can enter those documents from any terminal and print them on HP laser printers. More than 50 fonts are available, in point sizes from 5 to 72, with special mathematical and language symbols. The license fee is $1600, including a software device driver for a daisy-wheel printer; you'll pay $1600 more for the laser printer driver.

Tingler!, from TeXeT, gives information on run-time behavior of programs on the HP-3000. You can optimize and resegment programs based on this information; the information includes how many times each procedure and system intrinsic was called and where it was called from, how many intersegment transfers there were and where they occurred, and how much CPU time was expended in each procedure and intrinsic. Tingler! also detects logic errors and redundant code. It analyzes the program by patching itself into a copy of the executable program file, collecting performance data during a normal run of the program, and providing an off-line analysis—all without requiring recompilation of the program. License fee is $1000. **Reader Service No. 117.**

Go East, Young Programmer

One of the most interesting computer conferences of the year is likely to be **Computer China**, to be held in Xiamen Special Economic Zone in the Peoples' Republic of China from November 25 to December 1. This event is sanctioned by the Ministry of Electronics and backed by the China Microcomputer Applications Association, the China Research Society for Modernization of Management, and the China Instrument Society. **Reader Service No. 125.**

Another interesting conference, also to be held in the Orient, is billing itself as the first to be held on a ship. The **AFIPS-ASIA '85**—"The First Floating Shipboard Computer Expo"—is sponsored by the American Federation of Information Processing Societies (AFIPS). The ship, the *World Wide Expo*, has two large exhibition halls, five conference rooms, several international restaurants, and

private businessrooms. It leaves Tokyo on February 14, 1985, stops at Osaka and Kitakyushu (Japan), Taipei, and Hong Kong, and docks in Singapore on March 2. AFIPS, by the way, is comprised of the folks who bring you NCC and the Office Automation Conference (OAC). **Reader Service No. 127.**

IEEE Confers with ACM

Allow me to be the first to announce that the world's two largest professional societies for computer scientists and engineers, the Association for Computing Machinery (ACM) and the Institute of Electrical and Electronics Engineers (IEEE), will hold their first joint annual technical conference. Scheduled to be held in November, 1986, the gathering is to be called the **Fall Joint Computer Conference (FJCC)**. The two organizations have combined membership ofmore than 125,000.
Reader Service No. 123.

Contact Points

Association for Computing Machinery (ACM), 11 West 42nd St., New York, NY 10036; (212) 869-7440.

American Federation of Information Processing Societies (AFIPS), 1899 Preston White Drive, Reston, VA 22091; (703) 620-8900.

Charter Software, Box 70, Monticello, IL 61856; (217) 762-7143.

Computer China, c/o Kallman Associates, 5 Maple Court, Ridgewood, NJ 07450; (201) 652-7070.

Forth Interest Group, Box 1105, San Carlos, CA 94070; FIG Hot Line: (415) 962-8653.

Institute of Electrical and Electronics Engineers Computer Society, 1109 Spring St., Suite 300, Silver Spring, MD 20901; (301) 589-8142.

Muse Software, 347 North Charles Street, Baltimore, MD 21201; (301) 659-7212.

Peerless Engineering Service, 5819 Soquel Drive, Soquel, CA 95073; (408) 462-0330, or for orders, call (800) 824-7888, Operator 419.

Phoenix Software Associates, Ltd., 1420 Providence Highway, Suite 260, Norwood, MA 02062; (617) 769-7020.

Practicorp International Inc., The Silk Mill, 44 Oak Street, Newton Upper Falls, MA 02164, (617) 965-9870.

Rapitech Systems, Inc., 565 Fifth Ave., New York, NY 10017; (212) 687-6255.

Sabadia Export Corp., Box 1132, Yorba Linda, CA 92686; (714) 630-9335.

Software Architects, 1912 Grant, Berkeley, CA 94703; (415) 549-3185.

TeXet Company, 163 Linden Lane, Mill Valley, CA 94941; (415) 388-8853.

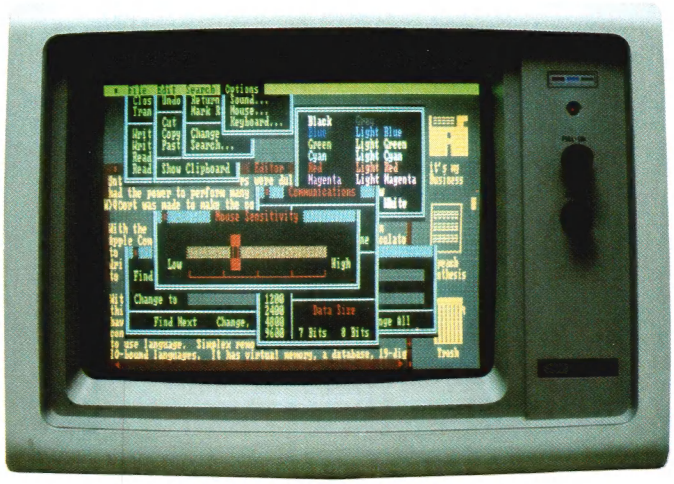
DDJ

Reader Ballot
Vote for your favorite feature/article.
Circle Reader Service **No. 198.**

We thought about calling it MacSimplex . . . after all it makes your IBM®PC behave like a Macintosh™ and much more . . .

and with over two years in the making, the Simplex Database Management System has features like 32-megabyte virtual memory and the most powerful networked/relational database in the microcomputer industry. Simplex was designed around how you think and the Macintosh way, so that you can use your favorite mouse to handle those mundane tasks like menu selection and data manipulation. And, if you don't have a mouse, you can use our keyboard mouse simulator, MouSim™.

Pop-up and pull-down menus, dialog and alert boxes are not just added features, they are the heart of the Simplex way. In addition, Simplex gives you both a software and a hardware floating point capability, each with 19-digit accuracy. It permits login, password, privilege, and can be used on a local area network. Simplex has full communications and a remote or local printer spooler. Above all, Simplex is modular and grows with you! Simplex also has a full-featured, English-like language which is simple to use.



You can't buy Simplex™, but it is now available as an integral part of *it's my* ***Business***™ and will be used by *it's my* ***Word***™, *it's my* ***Graphics***™, . . .

Businessmen! *it's my* ***Business*** will revolutionize the way that you handle your business. It saves time, money, and standardizes your system for all who use it. *it's my* ***Business*** comes with applications like accounting, interoffice or intraoffice mail, editing, invoicing, inventory managment, mail list, calendar, scheduler, forms and more. You can modify each of these to create applications specifically designed for you... maybe we should have called it "it's your Business".

Professionals! *it's my* ***Business*** has over 200 pages of examples and demonstrations to show you how to solve your everyday professional problems. And if these examples aren't enough, we give you a complimentary one-year subscription to Questalk™, our hands-on Simplex applications magazine.

System integrators and consultants, beware! If you are not using *it's my* ***Business*** with Simplex to solve your problems, don't be surprised when more novice programmers solve that complex math, industrial engineering, or business problem faster. We think that you can cut your concept-to-development time by an order of magnitude!

it's my ***Business*** (includes *it's my* ***Editor***) - $695.00
it's my ***Business*** Demo Disk - $20.00
it's my ***Editor*** - $100.00.

Quest Research software is available through your local computer store or through mail order from Quest Software Corporation at (205) 539-8086. 303 Williams Avenue, Huntsville, AL 35801.

Value added resellers and dealers please contact Quest Research, Incorporated at (800) 558-8088. 303 Williams Avenue, Huntsville, AL. 35801.

Quest™
Quest Research Inc.

This is THE PASCAL COMPILER You've Been Hearing About

"It's almost certainly better than IBM's Pascal for the PC... Recommended."
Jerry Pournelle
Byte, May 1984

$49.95

"If you don't have CP/M [for your Apple], Turbo Pascal is reason enough to buy it."
Cary Hara
Softalk Apple, May 1984

". . . an excellent product at an extraordinary price."
David D. Clark, *Dr. Dobb's Journal,* June 1984

And Now It's Even Better Than You've Heard!

- **Windowing (IBM PC, XT, jr. or true compatibles)**
- **Color, Sound and Graphics Support (IBM PC, XT, jr. or true compatibles)**
- **Optional 8087 Support *(available at an additional charge)***
- **Automatic Overlays**
- **A Full-Screen Editor that's even better than ever**
- **Full Heap Management—via dispose procedure**
- **Full Support of Operating System Facilities**
- **No license fees.** You can sell the programs you write with Turbo Pascal without extra cost.

Yes. We still include Microcalc . . . the sample spreadsheet written with Turbo Pascal. You can study the source code to learn how a spreadsheet is written . . . it's right on the disk.* And, if you're running Turbo Pascal with the 8087 option, you'll never have seen a spreadsheet calculate this fast before!

**Except Commodore 64 CP/M.*

Order Your Copy of TURBO PASCAL® VERSION 2.0 Today

For VISA and MasterCard orders call toll free: 1-800-255-8008
In California: 1-800-742-1133
(lines open 24 hrs, 7 days a week)

Dealer & Distributor Inquiries Welcome 408-438-8400

Choose One (please add $5.00 for shipping and handling for U.S. orders. Shipped UPS)

_____ Turbo Pascal 2.0 $49.95 + $5.00
_____ Turbo Pascal with 8087 support $89.95 + $5.00
_____ Update (1.0 to 2.0) Must be accompanied by the original master $29.95 + $5.00
_____ Update (1.0 to 8087) Must be accompanied by the original master $69.95 + $5.00

Check _____ Money Order ___
VISA _____ Master Card ___
Card #: _____
Exp. date: _____

BORLAND INTERNATIONAL
Borland International
4113 Scotts Valley Drive
Scotts Valley, California 95066
TELEX: 172373

My system is: 8 bit ___ 16 bit ___
Operating System: CP/M 80 ___
CP/M 86 ___ MS DOS ___ PC DOS ___
Computer: _____
Disk Format: _____
Please be sure model number & format are correct.
Name: _____
Address: _____
City/State/Zip: _____
Telephone: _____

California residents add 6% sales tax. Outside U.S.A. add $15.00 (If outside of U.S.A. payment must be by bank draft payable in the U.S. and in U.S. dollars.) Sorry, no C.O.D. or Purchase Orders. E11

Circle **no. 8** on reader service card.